Automatic parallelization of simulation code for equation-based models with software pipelining and measurements on three platforms

Conference Paper *in* ACM SIGARCH Computer Architecture News · November 2008

DOI: 10.1145/1556444.1556451 · Source: DBLP

CITATIONS
13

READS
122

4 authors, including:

Peter Fritzson
Linköping University
415 PUBLICATIONS **5,716** CITATIONS

SEE PROFILE

Christoph Kessler
Linköping University
198 PUBLICATIONS **1,611** CITATIONS

SEE PROFILE

Swedish Multicore Initiative

First Swedish Workshop on Multi-Core Computing MCC-08

November 27-28, 2008
Ronneby, Sweden

Editor: Håkan Grahn
Blekinge Institute of Technology, Sweden

Contents

Preface

Multicore processors have become the main computing platform for current and future computer systems. This calls for a forum to discuss the challenges and opportunities of both designing and using multicore systems. The objective of this workshop is to bring together researchers and practitioners from academia and industry to present and discuss the recent work in the area of multicore computing. The workshop is the first of its kind in Sweden, and it is co-organized by Blekinge Institute of Technology and the Swedish Multicore Initiative (http://www.sics.se/multicore/).

The technical program was put together by a distinguished program committee consisting of people from both from academia and industry in Sweden. We received 16 extended abstracts. Each abstract was sent to four members of the program committee. In total, we collected 64 review reports. The abstracts were judged based on their merits in terms of relevance to the workshop, significance and originality, as well as the scientific and presentation quality. Based on the reviews, the program committee decided to accept 12 papers for inclusion in the workshop, giving an acceptance rate of 75%. The accepted papers cover a broad range of topics, such as programming techniques and languages, compiler and library support, coherence and consistency issues, and verification techniques for multicore systems.

This workshop is the result of several people's effort. First of all, I would like to thank Monica Nilsson and Madeleine Rovegård for their help with many practical arrangements and organizational issues around the workshop. Then, I would like to thank the program committee for their dedicated and hard work, especially finishing all reviews on time despite the short time frame so we could send out author notifications as scheduled. Finally, I would like to thank the people in the steering committee for the Sweden Multicore Initiative for valuable and fruitful discussions about how to make this workshop successful.

With these words, I welcome you to the workshop!

Håkan Grahn
Organizer and Program Chair MCC-08
Blekinge Institute of Technology

Program committee

Mats Brorsson, Royal Institute of Technology
Jakob Engblom, Virtutech AB
Karl-Filip Faxén, Swedish Institute of Computer Science
Håkan Grahn, Blekinge Institute of Technology (program chair)
Erik Hagersten, Uppsala University
Per Holmberg, Ericsson AB
Sverker Janson, Swedish Institute of Computer Science
Magnus Karlsson, Enea AB
Christoph Kessler, Linköping University
Krzysztof Kuchcinski, Lund University
Björn Lisper, Mälardalen University
Per Stenström, Chalmers University of Technology
Andras Vajda, Ericsson Software Research

Workshop Program

Thursday 27/11

10.00 - 10.30 Registration etc
10.30 - 10.45 Welcome address
10.45 - 12.15 Paper session 1: Programming on specialized platforms

A Domain-specic Approach for Software Development on Multicore Platforms
Jerker Bengtsson and Bertil Svensson

On Sorting and Load-Balancing on GPUs
Daniel Cederman and Philippas Tsigas

Non-blocking Programming on Multi-core Graphics Processors
Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus

12.15 - 13.30 Lunch
13.30 - 14.30 Keynote speaker: Dr. Joakim M. Persson, Ericsson AB
14.30 - 15.00 Coffee break
15.00 - 16.30 Paper session 2: Language and compilation techniques

OpenDF - A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems
Shuvra S. Bhattacharyya, Gordon Brebner, Johan Eker, Jörn W. Janneck, Marco Mattavelli, Carl von Platen, and Mickael Raulet

Optimized On-Chip Pipelining of Memory-Intensive Computations on the Cell BE
Christoph W. Kessler and Jörg Keller

Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms
Håkan Lundvall, Kristian Stavåker, Peter Fritzson, and Christoph Kessler

19.00 Dinner

Friday 28/11

8.30 - 10.00 Paper session 3: Coherence and consistency

A Scalable Directory Architecture for Distributed Shared-Memory Chip Multiprocessors
Huan Fang and Mats Brorsson

State-Space Exploration for Concurrent Algorithms under Weak Memory Orderings
Bengt Jonsson

Model Checking Race-Freeness
Parosh Aziz Abdulla, Frédéric Haziza, and Mats Kindahl

10.00 - 10.30 Coffee break
10.30 - 12.00 Paper session 4: Library suppport for multicore computing

NOBLE: Non-Blocking Programming Support via Lock-Free Shared Abstract Data Types
Håkan Sundell and Philippas Tsigas

LFTHREADS: A lock-free thread library
Anders Gidenstam and Marina Papatriantafilou

Wool - A Work Stealing Library
Karl-Filip Faxén

12.00 Closing remarks
12.15 Lunch

Paper session 1: Programming on specialized platforms

A Domain-specific Approach for Software Development on Manycore Platforms

Jerker Bengtsson and Bertil Svensson
Centre for Research on Embedded Systems
Halmstad University
PO Box 823, SE-301 18 Halmstad, Sweden
Jerker.Bengtsson@hh.se

Abstract

The programming complexity of increasingly parallel processors calls for new tools that assist programmers in utilising the parallel hardware resources. In this paper we present a set of models that we have developed as part of a tool for mapping dataflow graphs onto manycores. One of the models captures the essentials of manycores identified as suitable for signal processing, and which we use as target for our algorithms. As an intermediate representation we introduce timed configuration graphs, which describe the mapping of a model of an application onto a machine model. Moreover, we show how a timed configuration graph by very simple means can be evaluated using an abstract interpretation to obtain performance feedback. This information can be used by our tool and by the programmer in order to discover improved mappings.

1. Introduction

To be able to handle the rapidly increasing programming complexity of manycore processors, we argue that *domain specific development tools are needed*. The signal processing required in radio base stations (RBS), see figure 1, is naturally highly parallel and described by computations on streams of data [9]. Each module in the figure encapsulates a set of functions, further exposing more pipeline-, data- and task level parallelism as a function of the number of connected users. Many radio channels have to be processed concurrently, each including fast and adaptive coding and decoding of digital signals. Hard real-time constraints imply that parallel hardware, including processors and accelerators is a prerequisite for coping with these tasks in a satisfactory manner.

One candidate technology for building baseband platforms is manycores. However, there are many issues that have to be solved regarding development of complex signal processing software for manycore hardware. One such is

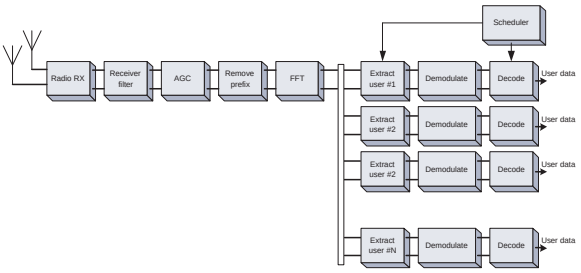


Figure 1. A simplified modular view of the principal functions of the baseband receiver in long term evolution (LTE) RBS.

the need for tools that reduce the programming complexity and abstract the hardware details of a particular manycore processor. We believe that if industry is to adopt manycore technology *the application software, the tools and the programming models need to be portable.*

Research has produced efficient compiler heuristics for programming languages based on streaming models of computation (MoC), achieving good speedup and high throughput for parallel benchmarks [3]. However, even though a compiler can generate optimized code the programmer is left with very little control of how the source program is transformed and mapped on the cores. This means that if the resulting code output does not comply with the system timing requirements, the only choice is to try to restructure the source program. We argue that *experienced application programmers must be able to direct and specialize the parallel mapping strategy by giving directive tool input.*

For complex real-time systems, such as baseband processing platforms, we see a need for tunable code parallelization- and mapping tools, allowing programmers to take the system's real-time properties into account during the optimization process. Therefore, complementary to

fully automatized parallel compilers, we are proposing an iterative code parallelization- and mapping tool flow that allows the programmer to tune mapping by:

- analyzing the result of a parallel code map using performance feedback
- giving timing constraints, clustering and core allocation directives as input to the tool

In our work we address the design and construction of one such tool. We focus on suitable well defined dataflow models of computation for modeling applications and manycore targets, as well as the base for our intermediate representation for manycore code-generation. One such model, synchronous dataflow (SDF), is very suitable for describing signal processing flows. It is also a good source for code-generation, given that it has a natural form of parallelism that is a good match to manycores. The goal of our work is a tool chain that allows the software developer to specify a manycore architecture (using our *machine model*), to describe the application (using SDF) and to obtain a generated mapping that can be evaluated (using our *timed configuration graph*). Such a tool allows the programmer to explore the run time behaviour of the system and to find successively better mappings. We believe that this iterative, machine assisted, workflow, is good in order to keep the application portable while being able to make trade-offs concerning throughput, latency and compliance with real-time constraint on different platforms.

In this paper we present our set of models and show how we can analyze the mapping of an application onto a manycore. More specifically, the contributions of this paper are as follows:

- A parallel machine model usable for modelling array-structured, tightly coupled manycore processors. The model is presented in Section 2, and in Section 3 we demonstrate modeling of one target processor.
- A graph-based intermediate representation (IR), used to describe a mapping of an application on a particular manycore in the form of a (*a timed configuration graph*). The use of this IR is twofold. We can perform an abstract interpretation that gives us feedback about the dynamic behaviour of the system. Also, we can use it to generate target code. We present the IR in Section 4.
- We show in Section 5 how parallel performance can be evaluated through abstract interpretation of the timed configuration graph. As a proof of concept we have implemented our interpreter in the Ptolemy II software framework using dataflow process networks.

We conclude our paper with a discussion of our achievements and future work.

2 Model Set

In this section we present the model set for constructing *timed configuration graph*s. First we discuss the application model, which describes the application processing requirements, and then the machine model, which is used to describe computational resources and performance of manycore targets.

2.1 Application Model

We model an application using SDF, which is a special case of a computation graph [5]. An SDF graph constitutes a network of actors - atomic or composite of variable granularity - which asynchronously compute on data distributed via synchronous uni-directional channels. By definition, actors in an SDF graph fire (compute) in parallel when there are enough tokens available on the input channels. An SDF graph is computable if there exists at least one static repetition schedule. A repetition schedule specifies in which order and how many times each actor is fired. If a repetition schedule exists, buffer boundedness and deadlock free execution is guaranteed. A more detailed description of the properties of SDF and how repetition schedules are calculated can be found in [6].

The Ptolemy II modelling software provides an excellent framework for implementing SDF evaluation- and code generator tools [1]. We can very well consider an application model as an executable specification. For our work, it is not the correctness of the implementation that is in focus. We are interested in analyzing the dynamic, non-functional behaviour of the system. For this we rely on measures like worst case execution time, size of dataflows, memory requirements etc. We assume that these data have been collected for each of the actors in the SDF graph and are given as a tuple

$$< r_p, r_m, R_s, R_r >$$

where

- r_p is the worst case computation time, in number of operations.
- r_m is the requirement on local data allocation, in words.
- $R_s = [r_{s_1}, r_{s_2}, ..., r_{s_n}]$ is a sequence where r_{s_i} is the number of words produced on channel i each firing.
- $R_r = [r_{r_1}, r_{r_2}, ..., r_{r_m}]$ is a sequence r_{r_j} is the number of words consumed on channel j each firing.

2.2 Machine Model

One of the early, reasonably realistic, models for distributed memory multiprocessors, is the LogP model [2]. Work has been done to to refine this model, for example taking into account hardware support for long messaging, and to capture memory hierarchies. A more recent parallel machine model for multicores, which considers different core granularities and requirements on on-chip and off-chip communication is Simplefit [7]. However, this model was derived with the purpose of exploring optimal grain size and balance between memory, processing, communication and global I/O, given a VLSI budget and a set of computation problems. Since it is not intended for modeling dynamic behaviour of a program, it does not include a fine-granular model of the communication. Taylor et al. propose a taxonomy (AsTrO) for comparison of scalar operand networks [11]. They also provide a tuple based model for comparing and evaluating performance sensitivity of on-chip network properties.

We propose a manycore machine model based on Simplefit and the AsTrO taxonomy, which allows a fairly fine-grained modeling of parallel computation performance including the overhead of operations associated with communication. The machine model comprises a set of parameters describing the computational resources and a set of abstract performance functions, which describe the computational performance of computations, communication and memory transactions. We will later show in Section 5 how we can can model dynamic, non-functional behavior of a dataflow graph mapped on a manycore target, by incorporating the machine model in a dataflow process network.

2.2.1 Machine Specification

We assume that cores are connected in a mesh structured network. Further that each core has individual instruction decoding capability and software managed memory load and store functionality, to replace the contents of core local memory. We describe the resources of such a manycore architecture using two tuples, M and F. M consists of a set of parameters describing the processors resources:

$$M =< (x, y), p, b_g, g_w, g_r, o, s_o, s_l, c, h_l, r_l, r_o >$$

where

- (x, y) is the number of rows and columns of cores.
- p is the processing power (instruction throughput) of each core, in *operations per clock cycle*.
- b_g is global memory bandwidth, in *words per clock cycle*
- g_w is the penalty for global memory write, in *words per clock cycle*
- g_r is the penalty for global memory read, in *words per clock cycle*
- o is software overhead for initiation of a network transfer, in *clock cycles*
- s_o is core send occupancy, in *clock cycles*, when sending a message.
- s_l is the latency for a sent message to reach the network, in *clock cycles*
- c is the bandwidth of each interconnection link, in *words per clock cycle*.
- h_l is network hop latency, in *clock cycles*.
- r_l is the latency from network to receiving core, in *clock cycles*.
- r_o is core receive occupancy, in *clock cycles*, when receiving a message

F is a set of abstract functions describing the performance of computations, global memory transactions and local communication:

$$F(M) =< t_p, t_s, t_r, t_c, t_{gw}, t_{gr} >$$

where

- t_p is a function evaluating the time to compute a list of instructions
- t_s is a function evaluating the core occupancy when sending a data stream
- t_r is a function evaluating the core occupancy when receiving a data stream
- t_c is a function evaluating network propagation delay for a data stream
- t_{gw} is a function evaluating the time for writing a stream to global memory
- t_{gr} is a function evaluating the time for reading a stream from global memory

A specifc manycore processor is modeled by giving values to the parameters of M and by defining the functions $F(M)$.

3 Modeling the RAW Processor

In this section we demonstrate how we configure our machine model in order to model the RAW processor [10]. RAW is a tiled, moderately parallel MIMD architecture with 16 programmable tiles, which are tightly connected via two different types of communication networks: two statically- and two dynamically routed. Each tile has a MIPS-type pipeline and is equipped with 32 KB of data and 96 KB instruction caches.

3.1 Parameter Settings

We are assuming a RAW setup with non-coherent off-chip global memory (four concurrently accessible DRAM banks), and that software managed cache mode is used. Furthermore, we concentrate on modeling usage of the dynamic networks, which are dimension-ordered, wormhole-routed, message-passing type of networks. The parameters of M for RAW with this configuration are as follows:

$$\begin{aligned} M =< \quad (x,y) &= (4,4), \\ p &= 1, \\ b_g &= 1, \\ g_w &= 1, \\ g_r &= 6, \\ o &= 2, \\ s_o &= 1, \\ s_l &= 1, \\ c &= 1, \\ h_l &= 1, \\ r_l &= 1, \\ r_o &= 1 > \end{aligned}$$

In our model, we assume a core instruction throughput of p operations per clock cycle. Each RAW tile has an eight-stage, single-issue, in-order RISC pipeline. Thus, we set $p = 1$. An uncertainty here is that in our current analyses, we cannot account for pipeline stalls due to dependencies between instructions having non-equal instruction latencies. We need to make further practical experiments, but we believe that this in general will be averaged out equally on cores and thereby not have too large effects on the estimated parallel performance.

There are four shared off-chip DRAMs connected to the four east-side I/O ports on the chip. The DRAMs can be accessed in parallel, each having a bandwidth of $b_g = 1$ words per clock cycle per DRAM. The penalty for a DRAM write is $g_w = 1$ cycle and correspondingly for read operation $g_r = 6$ cycles.

Since the communication patterns for dataflow graphs are known at compile time, message headers can be precomputed when generating the communication code. The overhead includes sending the header and possibly an address (when addressing off-chip memory). We therefore set $o = 2$ for header and address overhead when initiating a message.

The networks on RAW are mapped to the core's register files, meaning that after a header has been sent, the network can be treated as destination or source operand of an instruction. Ideally, this means that the receive and send occupancy is zero. In practice, when multiple input and output dataflow channels are merged and physically mapped on a single network link, data needs to be buffered locally. Therefore we model send and receive occupancy – for each word to be sent or received – by $s_o = 1$ and $r_o = 1$ respectively. The network hop-latency is $h_l = 1$ cycles per hop and the link bandwidth is $c = 1$. Furthermore, the send and receive latency is one clock cycle when injecting and extracting data to and from the network: $s_l = 1$ and $r_l = 1$ respectively.

3.2 Performance Functions

We have derived the performance functions by studying the hardware specification and by making small comparable experiments on RAW. We will now show how the performance functions for RAW are defined.

Compute The time required to process the fire code of an actor on a core is expressed as

$$t_p(r_p, p) = \left\lceil \frac{r_p}{p} \right\rceil$$

which is a function of the requested number of operations r_p and core processing power p. To r_p we count all instructions except those related to network send- and receive operations.

Send The time required for a core to issue a network send operation is expressed as

$$t_s(R_s, o, s_o) = \left\lceil \frac{R_s}{\mathit{framesize}} \right\rceil \times o + R_s \times s_o$$

Send is a function of the requested amount of words to be sent, R_s, the software overhead $o \in M$ when initiating a network transfer, and a possible send occupancy $s_o \in M$. The $framesize$ is a RAW specific parameter. The dynamic networks allow message frames of length within the interval $[0, 31]$ words. For global memory read and write operations, we use RAWs cache line protocol with $framesize = 8$ words per message. Thus, the first term of t_s captures the software overhead for the number of messages required to send the complete stream of data. For connected actors that are mapped on the same core, we can choose to map channels in local memory. In that case we set t_s to o zero time.

Receive The time required for a core to issue a network receive operation is expressed as

$$t_r(R_r, o, r_o) = \left\lceil \frac{R_r}{framesize} \right\rceil \times o + R_r \times r_o$$

The receive overhead is calculated in a similar way as the send overhead, except that parameters of the receiving core replace the parameters of the sending core.

Network Propagation Time Modeling shared resources accurately with respect to contention effects is very difficult. Currently, we assume that SDF graphs are mapped so that the communication will suffer from no or a minimum of contention. In the network propagation time, we consider a possible network injection- and extraction latency at the source and destination as well as the link propagation time. The propagation time is expressed as

$$t_c(R_s, d, s_l, h_l, r_l) = s_l + d \times h_l + n_{turns} + r_l$$

Network injection- and extraction latency is captured by s_l and r_l respectively. Further, the propagation time is dependent on the network hop latency h_l and the number of network hops d, which are determined from the source and destination coordinates as $|x_s - x_d| + |y_s - y_d|$. Routing turns add an extra cost of one clock cycle. This is captured by the value of n_{turns} which, similar to d, is calculated using the source and destination coordinates.

Streamed Global Memory Read Reading from global memory on the RAW machine requires first one send operation (the core overhead which is captured by t_s), in order to configure the DRAM controller and set the address of memory to be read. The second step is to issue a receive operation to receive the memory contents on that address. The propagation time when streaming data from global memory to the receiving core is expressed as

$$t_{gr} = r_l + d \times h_l + n_{turns}$$

Note that memory read penalty is not included in this expression. This is accounted for in the memory model included in the IR. This is further discussed in Section 4

Streamed Global Memory Write Similarly to the memory read operation, writing to global memory require two send operations: one for configuring the DRAM controller (set write mode and address) and one for sending the data to be stored. The time required for streaming data from the sending core to global memory is evaluated by

$$t_{gw} = s_l + d \times h_l + n_{turns}$$

Like in stream memory read, the memory write penalty is accounted for in the memory model.

4 Timed Configuration Graphs

In this section we describe our manycore intermediate representation (IR). We call the IR a *timed configuration graph* because the usage of the IR is twofold:

- Firstly, the IR is a graph representing an SDF application graph, after it has been clustered and partitioned for a specific manycore target. We can use the IR as input to a code generator, in order to configure each core as well as the interconnection network and plan global memory usage of a specific manycore target.
- Secondly, by introducing the notion of time in the graph, we can use the same IR as input to an abstract interpreter, in order to evaluate the dynamic behaviour of the application when executed on a specific manycore target. The output of the evaluator can be used either directly by the programmer or to extract information feedback to the tool for suggesting a better mapping.

4.1 Relations Between Models and Configuration Graphs

A *configuration graph* $G^A_M(V, E)$ describes an application A mapped on the abstract machine M. The set of vertices $V = P \cup B$ consists of cores $p \in P$ and global memory buffers $b \in B$. Edges $e \in E$ represent dataflow channels mapped onto the interconnection network. To obtain a G^A_M, the SDF for A is partitioned into subgraphs and each subgraph is assigned to a core in M. The edges of the SDF that end up in one subgraph are implemented using local memory in the core, so they do not appear as edges in G^A_M. The edges of the SDF that reach between subgraphs can be dealt with in two different ways:

1. A network connection between the two cores is used and this appears as an edge in G^A_M
2. Global memory is used as a buffer. In this case, a vertex b (and associated input- and output edges) is introduced between the two cores in G^A_M.

When G^A_M has been constructed, each $v \in V$ and $e \in E$ has been assigned computation times and communication delays, calculated using the parameters of M and the performance functions $F(M)$ introduced in Section 2.2. These annotations reflect the performance when computing the application A on the machine M. We will now discuss how we use A and M to configure the vertices, edges and then computational delays of G^A_M.

4.1.1 Vertices.

We distinguish between two types of vertices in G_M^A: *memory* vertices and *core* vertices. Introducing *memory* vertices allows us to represent global memory. A *memory* vertex can be specified by the programmer, for example to store initial data. More typically, *memory* vertices are automatically generated when mapping channel buffers in global memory.

For *core* vertices, we abstract the firing of an actor by means of a sequence S of abstract *receive*, *compute* and *send* operations:

$$S = t_{r_1}, t_{r_2} \ldots t_{r_n}, t_p, t_{s_1}, t_{s_2}, \ldots, t_{s_m}$$

The *receive* operation has a delay corresponding to the timing expression t_r, representing the time for an actor to receive data through a channel. The delay of a *compute* operation corresponds to the timing expression t_p, representing the time required to execute the computations of an actor when it fires. Finally, the *send* operation has a delay corresponding to the timing expression t_s, representing the time for an actor to send data through a channel.

For a *memory* type of vertex, we assign delays specified by g_r and g_w in the machine model to account for memory read- and write latencies respectively.

When building G_M^A, multiple channels sharing the same source and destination can be merged and represented by a single edge, treating them as a single block or stream of data. Thus, there is always only one edge $e_{i,j}$ connecting the pair (v_i, v_j). We add one *receive* operation and one *send* operation to the sequence S for each input and output edge respectively.

4.1.2 Edges.

Edges represent dataflow channels mapped onto the interconnection network. The weight w of an edge $e_{i,j}$ corresponds to the communication delay between vertex v_i and vertex v_j. The weight w depends on whether we map the channel as a point-to-point data stream over the network, or in shared memory using a *memory* vertex.

In the first case we assign the edge a weight corresponding to t_c. When a channel buffer is placed in global memory, we substitute the edge in A by a pair of input- and output edges connected to a *memory* actor. We illustrate this by Figure 2. We assign a delay of t_{gr} and t_{gw} to the input and output edges of the *memory* vertex.

Figure 3 shows an example of a simple A transformed to one possible G_M^A. A repetition schedule for A in this example is $3(2ABCD)E$. The repetition schedule should be interpreted as: actor A fires 6 times, actors B, C and D fire 3 times, and actor E 1 time. The firing of A is repeated indefinitly by this schedule. We use dashed lines for actors of A mapped and translated to S inside each core vertex of G_M^A. The feedback channel from C to B is mapped

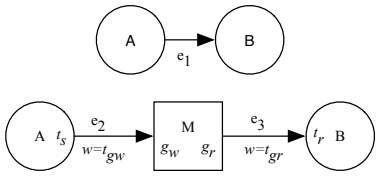


Figure 2. The lower graph (G_M^A) in the figure illustrates how the unmapped channel e_1, connecting actor A and actor B, in the upper graph (A), has been transformed and replaced by a global memory actor and edges e_2 and e_3.

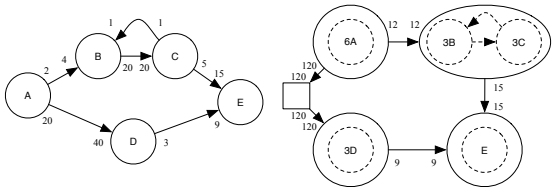


Figure 3. The graph to the right is one possible G_M^A for the graph A to the left.

in local memory. The edge from A to D is mapped via a global buffer and the others are mapped as point-to-point data streams. The integer values represent the send and receive rates of the channels (r_s and r_r), before and after A has been clustered and transformed to G_M^A, respectively. Note that these values in G_M^A are the values in A multiplied by the number of the repetition schedule.

5 Interpretation of Timed Configuration Graphs

In this section we show how we can make an abstract interpretation of the IR and how an interpreter can be implemented by very simple means on top of a dataflow process network. We have implemented such an interpreter using the dataflow process networks (PN) domain in Ptolemy. The PN domain in Ptolemy is a super set of the SDF domain. The main difference in PN, compared to SDF, is that PN processes fire asynchronously. If a process tries to read from an empty channel, it will block until there is new data available. The PN domain implemented in Ptolemy is a special case of Kahn process networks [4]. Unlike in a Kahn process network, PN channels have bounded buffer capacity, which implies that a process also temporarily blocks

when attempting to write to a buffer that is full [8]. This property makes it possible to easily model link occupancy on the network. Conclusively, a dataflow process network model perfectly mimics the behavior of the types of parallel hardware we are studying. Thus, a PN model is a highly suitable base for an intermediate abstraction for the processor we are targetting.

5.1 Parallel Interpretation using Process Networks

Each of the core and memory vertices of G_M^A is assigned to its own process. Each of the core and memory processes has a local clock, t, which iteratively maps the absolute start and stop time, as well as periods of blocking, to each operation in the sequence S.

A core process evaluates a vertex by means of a state machine. In each clock step, the current *state* is evaluated and then stored in the *history*. The *history* is a chronologically ordered list describing the *state* evolution from time $t = 0$.

5.2 Local Clocks

The clock t is process local and stepped by means of (not equal) time segments. The length of a time segment corresponds to the delay bound to a certain operation or the blocking time of a send or receive operation. The execution of send and receive operations in S is dependent on when data is available for reading or when a channel is free for writing, respectively.

5.3 States

For each vertex, we record during what segments of time computations and communication operations were issued, as well as periods where a core has been stalled due to send- and receive blocking. For each process, a *history* list maps to a state $type \in Stateset$, a start time t_{start} and a stop time t_{stop}. The $state$ of a vertex is a tuple

$$state =< type, t_{start}, t_{stop} >$$

The *StateSet* defines the set of possible state types:

$$StateSet = \{receive, compute, send, \\ blockedreceive, blockedsend\}$$

5.4 Clock Synchronisation

Send and receive are blocking operations. A read operation blocks until data is available on the edge and a write operation blocks until the edge is free for writing. During a time segment only one message can be sent over an edge. Clock synchronisation between communicating processes is managed by means of *event*s. Send and receive operations generate an *event* carrying a time stamp. An edge in G_M^A is implemented using channels having buffer size 1 (forcing write attempts on an occupied link to block), and a simple delay actor. It should be noted that each edge in A needs to be represented by a pair of opposite directed edges in G_M^A to manage synchronization.

receive($t_{receive}$)
  $t_{available}$ = get next send event from source vertex
  if($t_{receive} >= t_{available}$)
    $t_{read} = t_{receive+1}$
    $t_{blocked} = 0$
  else
    $t_{read} = t_{available+1}$
    $t_{blocked} = t_{available} - t_{receive}$
  end if
  put read event with time t_{read} to source vertex
  return $t_{blocked}$
end

Figure 4. Pseudo-code of the receive function. The get and put operations block if the event queue of the edge is empty or full, respectively.

5.4.1 Synchronised Receive

Figure 4 lists pseudo code of the blocking *receive* function. The value of the input $t_{receive}$ is the present time at which a receiving process issues a *receive* operation. The return value, $t_{blocked}$, is the potential blocking time. The time stamp $t_{available}$, is the time at which the message is available at the receiving core. If $t_{receive}$ is later or equal to $t_{available}$, the core immediately processes the receive operation and sets $t_{blocked}$ to 0. The *receive* function acknowledges by sending a read event to the sender, with the time stamp t_{read+1}. Note that a channel is free for writing as soon as the reciver has begun receiving the previous message. Also note that blocking time, due to unbalanced production and consumption rates, has been accounted for when analysing the timing expression for *send* and *receive* operations, T_s and T_r, as was discussed in Section 2.2. If $t_{receive}$ is earlier than $t_{available}$, the receiving core will block a number of clock cycles corresponding to $t_{blocked} = t_{available} - t_{receive}$.

5.4.2 Synchronised Send

Figure 5 lists pseudo code for the blocking *send* function. The value of t_{send} is the time at which the *send* operation was issued. The time stamp of the read event $t_{available}$ corresponds to the time at which the receiving vertex reads the previous message and thereby also when the edge is available for sending next message. If $t_{send} < t_{available}$, a *send* operation will block for $t_{blocked} = t_{available} - t_{send}$ clock cycles. Otherwise $t_{blocked}$ is set to 0. Note that all edges carrying receive events in the *configuration graph* must be initialised with a read event, otherwise interpretation will deadlock.

```
send(t_send)
  t_available = get read event from sink vertex
  if(t_send < t_available)
    t_blocked = t_available − t_send
  else
    t_blocked = 0
  end if
  put send event t_send + Δ_e + t_blocked to sink vertex
  return t_blocked
end
```

Figure 5. Pseudo-code of the send function. The value of Δ_e corresponds to the delay of the edge.

5.5 Vertex Interpretation

Figure 6 lists the pseudo code for interpretation of a vertex in G_M^A. It should be noted that, for space reasons, we have omitted to include the state code for global read and write operations. The function $interpretVertex()$ is finitely iterated by each process and the number of iterations, $iterations$, is equally set for all vertices when processes are initated. Each process has a local clock t and an operation counter op_cnt, both initially set to 0. The operations sequence S is a process local data structure, obtained from the vertex to be interpreted. Furthermore, each process has a list $history$ which initially is empty. Also, each process has a variable $curr\_oper$ which holds the currently processed operation in S.

The vertex interpreter makes state transitions depending on the current operation $curr\_oper$, the associated delay and whether *send* and *receive* operations block or not. As discussed in Section 5.4.1, the *send* and *receive* functions are the only blocking functions that can halt the interpretation in order to synchronise the clocks of the processes. The value of $t_{blocked}$ is set to the return value of *send* and *receive* when interpreting send and receive operations, respectively. The value of $t_{blocked}$ corresponds to the length of time a *send* or *receive* operation was blocked. If $t_{blocked}$ has a value > 0, a state of type *blocked_send* or *blocked_receive* is computed and added to the *history*.

```
interpretVertex()
  if(list S has elements)
    while(iterations > 0)
      get element op_cnt in S and put in curr_oper
      increment op_cnt

      if(curr_op is a Receive operation)
        set t_blocked = value of receive(t)
        if(t_blocked > 0)
          add state ReceiveBlocked(t, t_blocked) to hist.
          set t = t + t_blocked
        end if
        add state Receiving(t, Δ of curr_oper)
      end if

      else if(curr_op is a Compute operation)
        add state Computing(t, Δ of curr_oper)
      end if

      else if(curr_op is a Send operation)
        set t_blocked = value of send(t)
        if(t_blocked > 0)
          add state SendBlocked(t, t_blocked) to hist.
          set t = t + t_blocked
        end if
        add state Sending(t, Δ of curr_oper)
      end if

      if(op_cnt reached last index of S)
        set op_cnt = 0
        decrement iterations
        add state End(t) to history
      end if
      set t = t + Δ of curr_oper + 1
    end while
  end if
end
```

Figure 6. Pseudo-code of the interpretVertex function.

5.6 Model Calibration

We have implemented the abstract interpreter in the Ptolemy software modeling framework [1]. Currently, we have verified the correctness of the interpreter using a set of simple parallel computation problems from the literature. Regarding the accuracy of the model set, we have so far only compared the performance functions separately against corresponding operations on RAW. However, to evaluate and possibly tune the model for higher accuracy we need to do further experimental tests with different relevant signal processing benchmarks, especially including some more complex communication- and memory access patterns.

6 Discussion

We believe that tools supporting iterative mapping and tuning of parallel programs on manycore processors will play a crucial role in order to maximise application performance for different optimization criteria, as well as to reduce the parallel programming complexity. We also believe that using well defined parallel models of computation, matching the application, is of high importance in this matter.

In this paper we have presented our achievements towards the building of an iterative manycore code generation tool. We have proposed a machine model, which abstracts the hardware details of a specific manycore and provides a fine-grained instrument for evaluation of parallel performance. Furthermore, we have introduced and described an intermediate representation called *timed configuration graph*. Such a graph is annotated with computational delays that reflect the performance when the graph is executed on the manycore target. We have demonstrated how we compute these delays using the performance functions included in the machine model and the computational requirements captured in the application model. Moreover, we have in detail demonstrated how performance of a *timed configuration graph* can be evaluated using abstract interpretation.

As part of future work, we need to perform further benchmarking experiments in order to better determine the accuracy of our machine model compared to chosen target processors. Also, we have so far built *timed configuration graph*s by hand. We are especially interested in exploring tuning methods, using feedback information from the evaluator to set constraints in order to direct and improve the mapping of application graphs. Currently we are working on automatising the generation of the *timed configuration graph*s in our tool-chain, implemented in the Ptolemy II software modelling framework.

Acknowledgment

The authors would like to thank Henrik Sahlin and Peter Brauer at the Baseband Research group at Ericsson AB, Dr. Veronica Gaspes at Halmstad University, and Prof. Edward A. Lee and the Ptolemy group at UC Berkeley for valuable input and suggestions. This work has been funded by research grants from the Knowledge Foundation under the CERES contract.

References

[1] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2008-28, EECS Dept., University of California, Berkeley, Apr 2008.

[2] D. Culler, R. Karp, and D. Patterson. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of ACM SIGPLAN Symposium on Principles and Practices of Parallel programming*, May 1993.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in stream programs. In *Proc. of Twelfth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 5-10 1974. North-Holland Publishing Company.

[5] R. M. Karp and R. E. Miller. Properties of a Model for Parallel Computations:Determinancy, Termination, Queueing. *SIAM Journal of Applied Mathematics*, 14(6):1390–1411, November 1966.

[6] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Signal Processing. *IEEE Trans. on Computers*, January 1987.

[7] C. A. Moritz, D. Yeung, and A. Agarwal. SimpleFit: A Framework for Analyzing Design Tradeoffs in Raw Architectures. *IEEE Trans. on Parallel and Distributed Systems*, 12(6), June 2001.

[8] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, EECS Dept., University of California, Berkeley, Berkeley, CA, USA, 1995.

[9] H. Sahlin. Introduction and overview of LTE Baseband Algorithms. Powerpoint presentation, Baseband research group, Ericsson AB, February 2007.

[10] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[11] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar Operand Networks. *IEEE Trans. on Parallel and Distributed Systems*, 16(2):145–162, 2005.

On Sorting and Load-Balancing on GPUs

Daniel Cederman and Philippas Tsigas

Distributed Computing and Systems
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{cederman,tsigas}@chalmers.se

Abstract

In this paper we present GPU-Quicksort, an efficient Quicksort algorithm suitable for highly parallel multi-core graphics processors. Quicksort has previously been considered as an inefficient sorting solution for graphics processors, but we show that GPU-Quicksort often performs better than the fastest known sorting implementations for graphics processors, such as radix and bitonic sort. Quicksort can thus be seen as a viable alternative for sorting large quantities of data on graphics processors.

We also present a comparison of different load balancing schemes. To get maximum performance on the many-core graphics processors it is important to have an even balance of the workload so that all processing units contribute equally to the task at hand. This can be hard to achieve when the cost of a task is not known beforehand and when new sub-tasks are created dynamically during execution. With the recent advent of scatter operations and atomic hardware primitives it is now possible to bring some of the more elaborate dynamic load balancing schemes from the conventional SMP systems domain to the graphics processor domain.

1 Introduction

Multi-core systems are now commonly available on desktop systems and it seems very likely that in the future we will see an increase in the number of cores as both Intel and AMD targets many-core systems. But already now there are cheap and commonly available many-core systems in the form of modern graphics processors. Due to the many embarrassingly parallel problems in 3D-rendering, the graphics processors have come quite a bit on the way to massive parallelism and high end graphics processors currently boasts up to 240 processing cores.

Until recently the only way to take advantage of the GPU was to transform the problem into the graphics domain and use the tools available there. This however was a very awkward abstraction layer and made it hard to use. Better tools are now available and among these are CUDA, which is NVIDIA's initiative to bring general purpose computation to their graphics processors [5]. It consists of a compiler and a run-time for a C/C++-based language which can be used to create kernels that can be executed on CUDA-enabled graphics processors.

2 System Model

In CUDA you have unrestricted access to the main graphics memory, known as the *global memory*. There is no cache memory but the hardware supports coalescing memory operations so that several read operations on consecutive memory locations can be merged into one big read or write operation which will make better use of the memory bus and provide far greater performance. Newer graphics processors support most of the common atomic operations such as CAS (Compare-And-Swap) and FAA (Fetch-And-Add) when accessing the memory and these can be used to implement efficient parallel data structures.

The high range graphics processors currently consist of up to 32 multiprocessors each, which can perform SIMD (Single Instruction, Multiple Data) instructions on 8 memory positions at a time. Each multiprocessor has 16 kB of a very fast local memory that allows information to be communicated between threads assigned to the same *thread block*. A thread block is a set of threads that are assigned to run at the same multiprocessors. All thread blocks have the same number of threads assigned to them and this number is specified by the programmer. Depending on how many

The results presented in this extended abstract appeared before in the Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008), Lecture Notes in Computer Science Vol.: 5193, Springer-Verlag 2008 [2] and in the Proceedings of the 11th Graphics Hardware (GH 2008), ACM/Eurographics Association 2008 [3].

registers and how much local memory the block of threads requires, there could be multiple blocks assigned to a single multiprocessor. All the threads in a scheduled thread block are run from start to finish before the block can be swapped out, so if more blocks are needed than there is room for on any of the multiprocessors, the leftover blocks will be run sequentially.

The GPU schedules threads depending on which *warp* they are in. Threads with *id* 0..31 are assigned to the first warp, threads with *id* 32..63 to the next and so on. When a warp is scheduled for execution, the threads which perform the same instructions are executed concurrently (limited by the size of the multiprocessor) whereas threads that deviate are executed sequentially.

3 Overview

Having a relatively parallel algorithm it is possible to get really good performance out of CUDA. It is important however to try to make all threads in the same warp perform the same instructions most of the time so that processor can fully utilize the SIMD operations and also, since there is no cache, to try to organize data so that memory operations coalesce as much as possible, something which is not always trivial. The local memory is very fast, just as fast as accessing a register, and should be used for common data and communication, but it is very small since it's being shared by a larger number of threads and there is a challenge in how to use it optimally.

This paper is divided into two parts. In the first part we present our Quicksort algorithm for graphics processors and in the second we present a comparison between different load balancing schemes.

4 GPU-Quicksort

We presented an efficient parallel algorithmic implementation of Quicksort, GPU-Quicksort, designed to take advantage of the highly parallel nature of graphics processors (GPUs) and their limited cache memory [2]. Quicksort has long been considered as one of the fastest sorting algorithms in practice for single processor systems, but until now it has not been considered as an efficient sorting solution for GPUs . We show that GPU-Quicksort presents a viable sorting alternative and that it can outperform other GPU-based sorting algorithms such as GPUSort and radix sort, considered by many to be two of the best GPU-sorting algorithms. GPU-Quicksort is designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed. It achieves this by using a two-phase design to keep the inter-thread synchronization low and by steering the threads so that their memory read operations are performed coalesced. It can also take advantage of the atomic synchronization primitives found on newer hardware, when available, to further improve its performance.

5 The Algorithm

The following subsection gives an overview of GPU-Quicksort. Section 5.2 will then go into the algorithm in more detail.

5.1 Overview

The method used by the algorithm is to recursively *partition* the sequence to be sorted, i.e. to move all elements that are lower than a specific pivot value to a position to the left of the pivot and to move all elements with a higher value to the right of the pivot. This is done until the entire sequence has been sorted.

In each partition iteration a new pivot value is picked and as a result two new subsequences are created that can be sorted independently. After a while there will be enough subsequences available that each thread block can be assigned one of them. But before that point is reached, the thread blocks need to work together on the same sequences. For this reason, we have divided up the algorithm into two, albeit rather similar, phases.

First Phase In the first phase, several thread blocks might be working on different parts of the same sequence of elements to be sorted. This requires appropriate synchronization between the thread blocks, since the results of the different blocks need to be merged together to form the two resulting subsequences.

Newer graphics processors provide access to atomic primitives that can aid somewhat in this synchronization, but they are not yet available on the high-end graphics processors. Because of that, there is still a need to have a thread block barrier-function between the partition iterations.

The reason for this is that the blocks might be executed sequentially and we have no way of knowing in which order they will be executed. The only way to synchronize thread blocks is to wait until all blocks have finished executing. Then one can assign new subsequences to them. Exiting and reentering the GPU is not expensive, but it is also not delay-free since parameters need to be copied from the CPU to the GPU, which means that we want to minimize the number of times we have to do that.

When there are enough subsequences so that each thread block can be assigned its own subsequence, we enter the second phase.

Second Phase In the second phase, each thread block is assigned its own subsequence of input data, eliminating the need for thread block synchronization. This means that the

second phase can run entirely on the graphics processor. By using an explicit stack and always recurse on the smallest subsequence, we minimize the shared memory required for bookkeeping.

Hoare suggested in his paper [9] that it would be more efficient to use another sorting method when the subsequences are relatively small, since the overhead of the partitioning gets too large when dealing with small sequences. We decided to follow that suggestion and sort all subsequences that can fit in the available local shared memory using an alternative sorting method.

In-place On conventional SMP systems it is favorable to perform the sorting in-place, since that gives good cache behavior. But on GPUs, because of their limited cache memory and the expensive thread synchronization that is required when hundreds of threads need to communicate with each other, the advantages of sorting in-place quickly fades away. Here it is better to aim for reads and writes to be coalesced to increase performance, something that is not possible on conventional SMP systems. For these reasons it is better, performance-wise, to use an auxiliary buffer instead of sorting in-place.

So, in each partition iteration, data is read from the primary buffer and the result is written to the auxiliary buffer. Then the two buffers switch places, with the primary becoming the auxiliary and vice versa.

5.1.1 Partitioning

The principle of two phase partitioning is outlined in Figure 1. The sequence to be partitioned is selected and it is then logically divided into m equally sized sections (Step a), where m is the number of thread blocks available. Each thread block is then assigned a section of the sequence (Step b).

The thread block goes through its assigned data, with all threads in the block accessing consecutive memory so that the reads can be coalesced. This is important, since reads being coalesced will significantly lower the memory access time.

Synchronization The objective is to partition the sequence, i.e. to move all elements that are lower than the pivot to a position to the left of the pivot in the auxiliary buffer and to move the elements with a higher value than the pivot to the right of the pivot. The problem here is to synchronize this in an efficient way. How do we make sure that each thread knows where to write in the auxiliary buffer?

Cumulative Sum A possible solution is to let each thread read an element and then synchronize the threads using a barrier function. By calculating a cumulative sum of the number of threads that want to write to the left and to the right of the pivot respectively, each thread would know that x threads with a lower thread id than its own are going to write to the left of the pivot and that y threads are going to write to the right of the pivot. Each thread then knows that it can write its element to either buf_{x+1} or $buf_{n-(y+1)}$, depending on if the element is higher or lower than the pivot.

A Two-Pass Solution But calculating a cumulative sum is not free, so to improve performance we go through the sequence two times. In the first pass each thread just counts the number of elements it has seen that have value higher (or lower) than the pivot (Step c). Then when the block has finished going through its assigned data, we use these sums instead to calculate the cumulative sum (Step d). Now each thread knows how much memory the threads with a lower id than its own needs in total, turning it into an implicit memory-allocation scheme that only needs to run once for every thread block, in each iteration.

In the first phase, where we have several thread blocks accessing the same sequence, an additional cumulative sum need to be calculated for the total memory used by each thread block (Step e).

When each thread knows where to store its elements, we go through the data in a second pass (Step g), storing the elements at their new position in the auxiliary buffer. As a final step, we store the pivot value at the gap between the two resulting subsequences (Step h). The pivot value is now at its final position which is why it doesn't need to be included in any of the two subsequences.

5.2 Detailed Description

5.2.1 The First Phase

The goal of the first phase is to divide the data into a large enough number of subsequences that can be sorted independently.

Work Assignment In the ideal case, each subsequence should be of the same size, but that is often not possible, so it is better to have some extra subsequences and let the scheduler balance the workload. Based on that observation, a good way to partition is to only partition subsequences that are longer than $minlength = n/maxseq$ and to stop when we have $maxseq$ number of subsequences.

In the beginning of each iteration, all subsequences that are larger than the $minlength$ are assigned thread blocks relative to their size. In the first iteration, the original subsequence will be assigned all available thread blocks. The subsequences are divided so that each thread block gets an equally large section to sort, as can be seen in Figure 1 (Step a and b).

First Pass When a thread block is executed on the GPU, it will iterate through all the data in its assigned sequence. Each thread in the block will keep track of the number of elements that are greater than the pivot and the number of elements that are smaller than the pivot. The data is read in chunks of T words, where T is the number of threads in

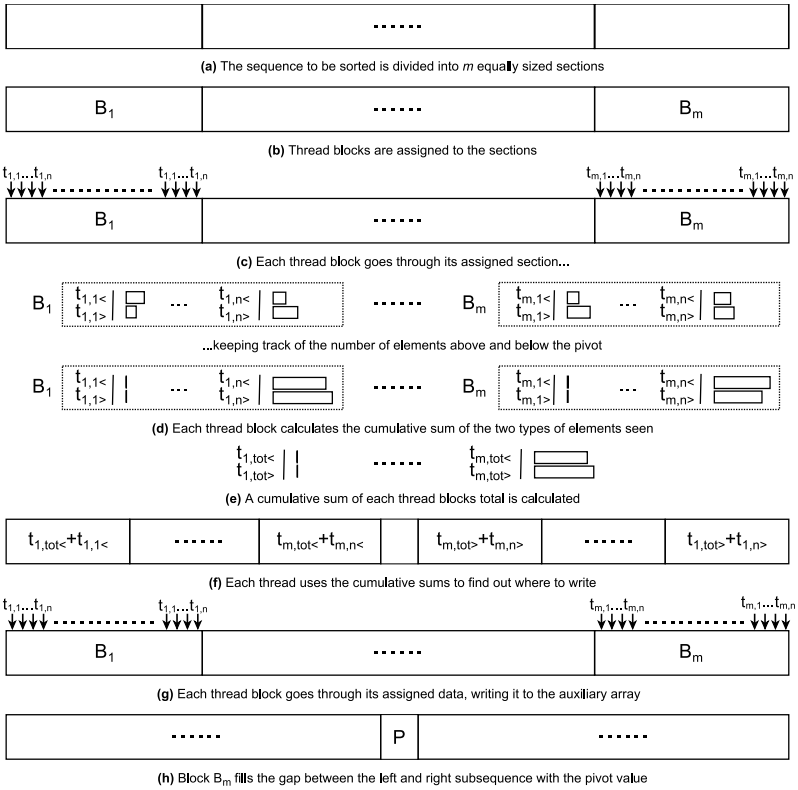


Figure 1. Partitioning a sequence (m thread blocks with n threads each).

each thread block. The threads read consecutive words so that the reads coalesce as much as possible.

Space Allocation Once we have gone through all the assigned data, we calculate the cumulative sum of the two arrays. We then use the atomic FAA-function to calculate the cumulative sum for all blocks that have completed so far. This information is used to give each thread a place to store its result, as can be seen in Figure 1 (Step c-f).

FAA is as of the time of writing not available on all GPUs. An alternative, if one wants to run the algorithm on the older, high-end graphics processors, is to divide the kernel up into two kernels and do the block cumulative sum on the CPU instead. This would make the code more generic, but also slightly slower on new hardware.

Second Pass Using the cumulative sum, each thread knows where to write elements that are greater or smaller than the pivot. Each block goes through its assigned data again and writes it to the correct position in the current auxiliary array. It then fills the gap between the elements that are greater or smaller than the pivot with the pivot value. We now know that the pivot values are in their correct final position, so there is no need to sort them anymore. They are therefore not included in any of the newly created subsequences.

Are We Done? If the subsequences that arise from the partitioning are longer than $minlength$, they will be partitioned again in the next iteration, provided we don't already have more than $maxseq$ subsequences. If we do have more than $maxseq$ subsequences, the next phase begins. Otherwise we go through another iteration. (See Algorithm 1).

5.2.2 The Second Phase

When we have acquired enough independent subsequences, there is no longer any need for synchronization between blocks. Because of this, the entire phase two can be run on the GPU entirely. There is however still the need for synchronization between threads, which means that we will use the same method as in phase one to partition the data. That is, we will count the number of elements that are greater or smaller than the pivot, do a cumulative sum so that each thread has its own location to write to and then move all elements to their correct position in the auxiliary buffer.

Stack To minimize the amount of fast local memory used, there is a very limited supply of it, we always recurse on the smallest subsequence. By doing that, Hoare have showed [9] that the maximum recursive depth can never go below $\log_2(n)$. We use an explicit stack as suggested by Hoare and implemented by Sedgewick, always storing the smallest subsequence at the top [12].

Overhead When a subsequence's size goes below a certain threshold, we use an alternative sorting method on it. This was suggested by Hoare since the overhead of Quicksort gets too big when sorting small sequences of data. When a subsequence is small enough to be sorted entirely in the fast local memory, we could use any sorting method that can be made to sort in-place, doesn't require much expensive thread synchronization and performs well when the number of threads approaches the length of the sequence to be sorted.

6 Experimental Evaluation

We ran the experiments on a dual-processor dual-core AMD Opteron 1.8GHz machine. Two different graphics processors were used, the low-end NVIDIA 8600GTS 256MiB with 4 multiprocessors and the high-end NVIDIA 8800GTX 768MiB with 16 multiprocessors. Since the 8800GTX provides no support for the atomic FAA operation we instead used an implementation of the algorithm that exits to the CPU for block-synchronization.

We compared GPU-Quicksort to the following state-of-the-art GPU sorting algorithms:

GPUSort Uses bitonic merge sort [6].

Radix-Merge Uses radix sort to sort blocks that are then merged [7].

Global Radix Uses radix sort on the entire sequence [13].

Hybridsort Uses a bucket sort followed by a merge sort [16].

STL-Introsort This is the Introsort implementation found in the C++ Standard Library. Introsort is based on Quicksort, but switches to heap-sort when the recursion depth gets too large. Since it is highly dependent on the computer system and compiler used, we only included it to give a hint as to what could be gained by sorting on the GPU instead of on the CPU [11].

We could not find an implementation of the Quicksort algorithm used by Sengupta et al., but they claim in their paper that it took over 2 seconds to sort 4M uniformly distributed elements on a 8800GTX [13].

We only measured the actual sorting phase, we did not include in the result the time it took to setup the data structures and to transfer the data on and off the graphics memory. The reason for this is the different methods used to transfer data which wouldn't give a fair comparison between the GPU-based algorithms. Transfer times are also irrelevant if the data to be sorted are already available on the GPU. Because of those reasons, this way of measuring has become a standard in the literature.

On the 8800GTX we used 256 thread blocks, each block having 256 threads. When a subsequence dropped below 1024 elements in size, we sorted it using bitonic sort. On the 8600GTS we lowered the amount of thread blocks to 128 since it has fewer multiprocessors. All implementations were compiled with the -O3 optimization flag.

We used different pivot selection schemes for the two phases. In the first phase we took the average of the minimum and maximum element in the sequence and in the second we picked the median of the first, middle and last element as the pivot, a method suggested by Singleton[15].

The source code of GPU-Quicksort is available for non-commercial use [4].

For benchmarking we used a uniform, sorted, zero, bucket, gaussian and staggered distribution which are defined and motivated in [8]. These are commonly used yardsticks to compare the performance of different sorting algorithms. The source of the random uniform values is the Mersenne Twister [10].

6.1 Discussion

Quicksort has a worst case scenario complexity of $O(n^2)$, but in practice, and on average when using a random pivot, it tends to be close to $O(n \log(n))$, which is the lower bound for comparison sorts. In all our experiments GPU-Quicksort has shown the best performance or been among the best. There was no distribution that caused problems to the performance of GPU-Quicksort. As can be seen when comparing the performance on the two GPUs, GPU-Quicksort shows a speedup of approximately 3 on the higher-end GPU. The higher-end GPU has a memory bandwidth that is 2.7 times higher and has four times the number of multiprocessors, indicating that the algorithm is bandwidth bound and not computation bound, which was the case with the Quicksort in [13].

On the CPU, Quicksort is normally seen as a faster algorithm as it can potentially pick better pivot points and doesn't need an extra check to determine when the sequence is fully sorted. The time complexity of radix sort is $O(n)$, but that hides a potentially high constant which is dependent on the key size. Optimizations are possible to lower this constant, such as constantly checking if the sequence has been sorted, but that can be expensive when dealing with longer keys. Quicksort being a comparison sort also means that it is easier to modify it to handle different key types.

The hybrid approach uses atomic instructions that were only available on the 8600GTS. We can see that it outperforms both GPU-Quicksort and the global radix sort on the uniform distribution. But it loses speed on the staggered distributions and becomes immensely slow on the zero distribution. The authors state that the algorithm drops in performance when faced with already sorted data, so they suggest randomizing the data first, but this wouldn't affect the result in the zero distribution.

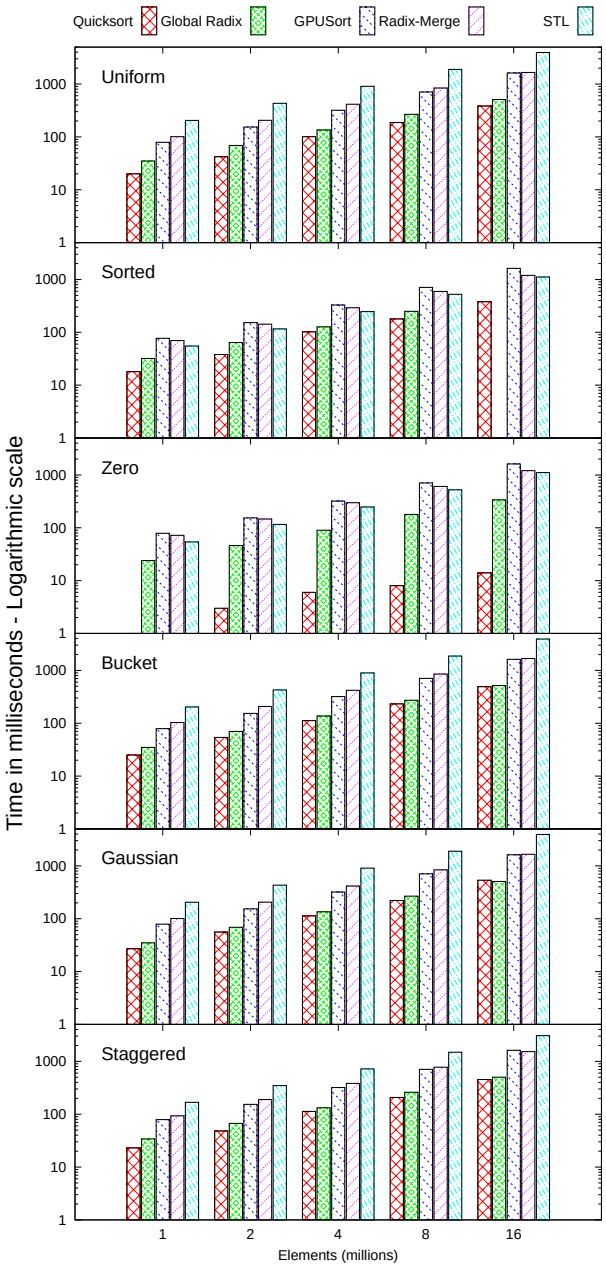


Figure 2. Results on the 8800GTX.

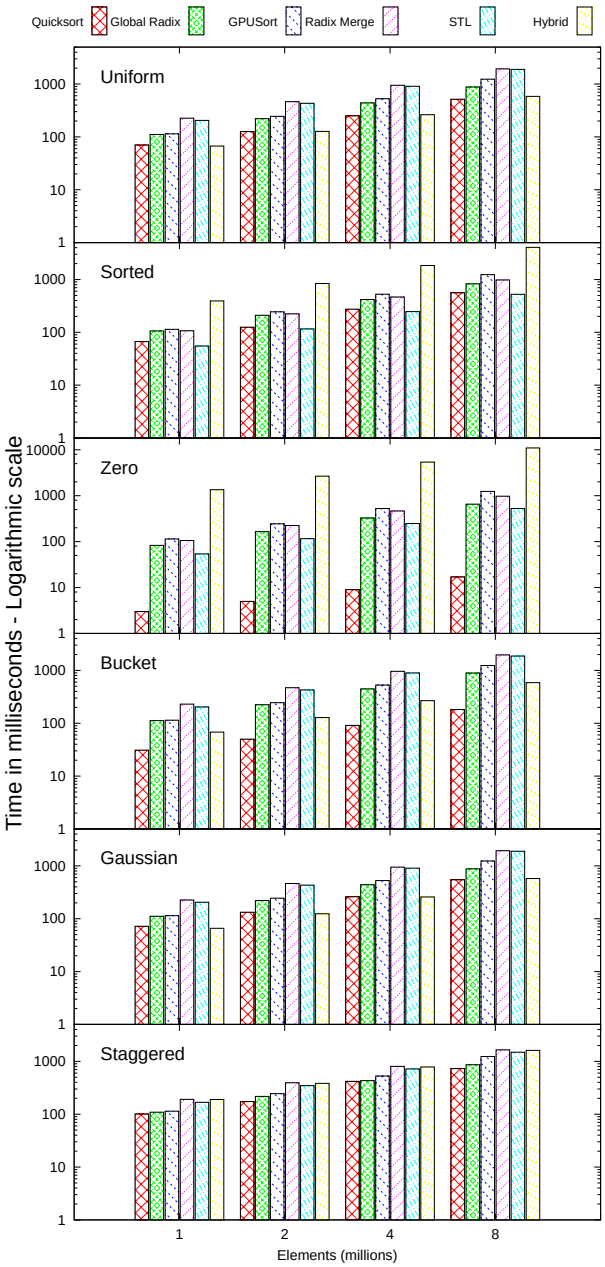


Figure 3. Results on the 8600GTS.

GPUSort doesn't increase as much in performance as the other algorithms when executed on the higher-end GPU. This is an indication that the algorithm is more computationally bound than the other algorithms. It goes from being much faster than the slow radix-merge to perform on par with and even a bit slower than it. The global radix sort showed a 3x speed improvement, as did GPU-Quicksort.

All algorithms showed about the same performance on the uniform, bucket and Gaussian distributions. GPUSort always shows the same result independent of distributions since it is a sorting network, which means it always performs the same number of operations regardless of the distribution. The staggered distribution was more interesting. On the low-end GPU the hybrid sorting was more than twice as slow as on the uniform distribution. GPU-Quicksort also dropped in speed and started to show the same performance as GPUSort. This can probably be attributed to the choice of pivot selection which was more optimized for uniform distributions. The zero distribution, which can be seen as an already sorted sequence, affected the algorithms to dif-

ferent extent. The STL reference implementation increased dramatically in performance since its two-way partitioning function always returned even partitions regardless of the pivot chosen. GPU-Quicksort shows the best performance as it does a three-way partitioning and can sort the sequence in $O(n)$ time.

7 Dynamic Load Balancing on Graphics Processors

Load balancing play a significant role in the design of efficient parallel algorithms and applications. In a step towards understanding the new dimensions of the problem that are introduced from the new graphics processors' features and capabilities, we have designed and compared four different dynamic load balancing methods to see which one is most suited to the highly parallel world of graphics processors [3]. Three of these methods were lock-free and one was lock-based. We evaluated them on the task of creating an octree partitioning of a set of particles. The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors features and capabilities might be required. They also showed that lock-free methods achieves better performance than blocking and that they can be made to scale with increased numbers of processing units.

8 Load Balancing Methods

This section gives an overview of the different load balancing methods we have compared in this paper.

8.1 Static Task List

The default method for load balancing used in CUDA is to divide the data that is to be processed into a list of blocks or tasks. Each processing unit then takes out one task from the list and executes it. When the list is empty all processing units stop and control is returned to the CPU.

This is a lock-free method and it is excellent when the work can be easily divided into chunks of similar processing time, but it needs to be improved upon when this information is not known beforehand. Any new tasks that are created during execution will have to wait until all the statically assigned tasks are done, or be processed by the thread block that created them, which could lead to an unbalanced workload on the multiprocessors.

The method, as implemented in this work, consists of two steps that are performed iteratively. The only data structures required are two arrays containing tasks to be processed and a tail pointer. One of the arrays is called the in-array and only allows read operations while the other, called the out-array, only allows write operations.

In the first step of the first iteration the in-array contains all the initial tasks. For each task in the array a thread block is started. Each thread block then reads task i, where i is the thread block ID. Since no writing is allowed to this array, there is no need for any synchronization when reading.

If any new task is created by a thread block while performing its assigned task, it is added to the out-array. This is done by incrementing the tail pointer using the atomic FAA-instruction. FAA returns the value of a variable and increments it by a specified number atomically. Using this instruction the tail pointer can be moved safely so that multiple thread blocks can write to the array concurrently.

The first step is over when all tasks in the in-array has been executed. In the second step the out-array is checked to see if it is empty or not. If it is empty the work is completed. If not, the pointers to the out- and in-array are switched so that they change roles. Then a new thread block for each of the items in the new in-array is started and this process is repeated until the out-array is empty.

8.2 Blocking Dynamic Task Queue

In order to be able to add new tasks during runtime we designed a parallel dynamic task queue that thread blocks can use to announce and acquire new tasks.

As several thread blocks might try to access the queue simultaneously it is protected by a lock so that only one thread block can access the queue at any given time. This is a very easy and standard way to implement a shared queue, but it lowers the available parallelism since only one thread block can access the queue at a time, even if there is no conflict between them.

The queue is array-based and uses the atomic CAS (Compare-And-Swap) instruction to set a lock variable to ensure mutual exclusion. When the work is done the lock variable is reset so that another thread block might try to grab it.

8.3 Lock-free Dynamic Task Queue

A lock-free implementation of a queue was implemented to avoid the problems that comes with locking and also in order to study the behavior of lock-free synchronization on graphics processors. A lock-free queue guarantees that, without using blocking at all, at least one thread block will always succeed to enqueue or dequeue an item at any given time even in presence of concurrent operations. Since an operation will only have to be repeated at an actual conflict it can deliver much more parallelism.

The implementation is based upon the simple and efficient array-based lock-free queue described in a paper by Tsigas and Zhang [17]. A tail pointer keeps track of the tail of queue and tasks are then added to the queue using

CAS. If the CAS-operation fails it must be due to a conflict with another thread block, so the operation is repeated on the new tail until it succeeds. This way at least one thread block is always assured to successfully enqueue an item.

The queue uses lazy updating of the tail and head pointers to lower contention. Instead of changing the head/tail pointer after every enqueue/dequeue operation, something that is done with expensive CAS-operations, it is only updated every x:th time. This increases the time it takes to find the actual head/tail since several queue positions needs to be checked. But by reading consecutive positions in the array, the reads will be coalesced by the hardware into one fast read operation and the extra time can be made lower than the time it takes to try to update the head/tail pointer x times.

8.4 Task Stealing

Task stealing is a popular load balancing scheme. Each processing unit is given a set of tasks and when it has completed them it tries to steal a task from another processing unit which has not yet completed its assigned tasks. If a unit creates a new task it is added to its own local set of tasks.

One of the most used task stealing methods is the lock-free scheme by Arora et al. [1] with multiple array-based double ended queues (*deques*). This method will be referred to as ABP task stealing in the remainder of this paper.

In this scheme each thread block is assigned its own deque. Tasks are then added and removed from the tail of the deque in a LIFO manner. When the deque is empty the process tries to steal from the head of another process' deque.

Since only the owner of the deque is accessing the tail of the deque there is no need for expensive synchronization when the deque contains more than one element. Several thread blocks might however try to steal at the same time, requiring synchronization, but stealing is assumed to occur less often than a normal local access. The implementation is based on the basic non-blocking version by Arora et al. [1]. The stealing is performed in a global round robin fashion, so thread block i looks at thread block $i + 1$ followed by $i + 2$ and so on.

9 Octree Partitioning

To evaluate the dynamical load balancing methods described in the previous section, they are applied to the task of creating an octree partitioning of a set of particles [14]. An octree is a tree-based spatial data structure that recursively divides the space in each direction, creating eight octants. This is done until an octant contains less than a specific number of particles. The fact that there is no information beforehand on how deep each branch in the tree will be, makes this a suitable problem for dynamic load balancing.

10 Experimental Evaluation

Two different graphics processors were used in the experiments, the 9600GT 512MiB NVIDIA graphics processor with 64 cores and the 8800GT 512MiB NVIDIA graphics processor with 112 cores.

We used two input distributions, one where all particles were randomly picked from a cubic space and one where they were randomly picked from a space shaped like a geometrical tube.

All methods where initialized by a single iteration using one thread block. The maximum number of particles in any given octant was set to 20 for all experiments.

10.1 Discussion

Figure 4 shows the time it took to partition two different particle sets on the 8800GT graphics processors using each of the load balancing methods while varying the number of threads per block and blocks per grid. The static method always uses one block per task and is thus shown in a 2D graph.

Figure 5 shows the time taken to partition particle sets of varying size using the combination of threads per block and blocks per grid found to be optimal in the previously described graph.

Figure 4 (a) clearly shows that using less than 64 threads with the blocking method gives us the worst performance in all of the experiments. This is due to the expensive spinning on the lock variable. These repeated attempts to acquire the lock causes the bus to be locked for long amounts of times during which only 32-bit memory accesses are done. With more than 64 threads the number of concurrent thread blocks is lowered from three to one, due to the limited number of available registers per thread, which leads to less lock contention and much better performance. This shows that the blocking method scales very poorly. In fact, we get the best result when using less than ten blocks, that is, by not using all of the multiprocessors!

The non-blocking queue-based method, shown in Figure 4 (b), can take better advantage of an increased number of blocks per grid. We see that the performance increases quickly when we add more blocks, but after around 20 blocks the effect fades. It was expected that this effect would be visible until we increased the number of blocks beyond 42, the number of blocks that can run concurrently when using less than 64 threads. This means that even though its performance is much better than the its blocking counterpart, it still does not scale as well as we would have wanted. This can also clearly be seen when we pass the

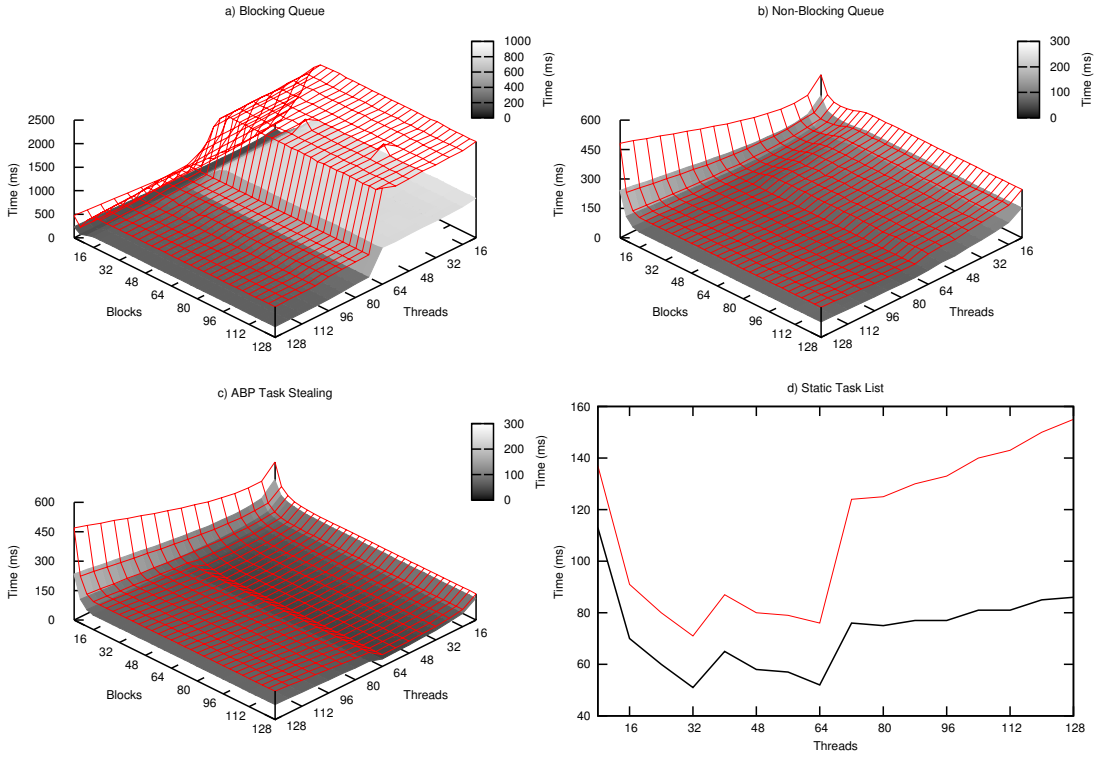


Figure 4. Comparison of load balancing methods on the 8800GT. Shows the time taken to partition a Uniform (filled grid) and Tube (unfilled grid) distribution of half a million particles using different combinations of threads/block and blocks/grid.

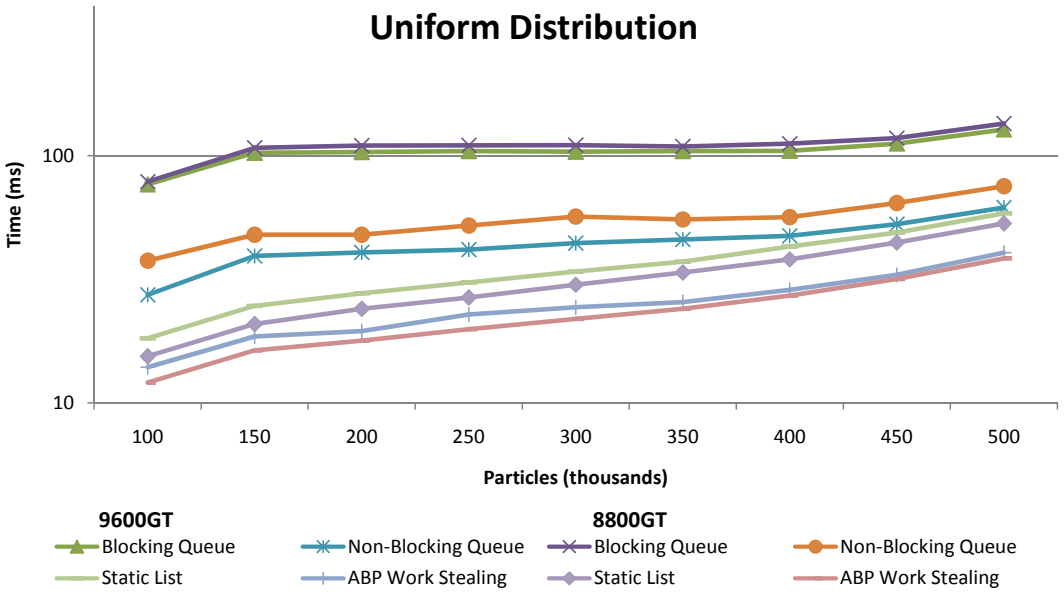


Figure 5. A comparison of the load balancing methods on the uniform distribution.

64 thread boundary and witness an increase in performance instead of the anticipated drop.

In Figure 4 (c) we see the result from the ABP task stealing and it lies more closely to the ideal. Adding more blocks increases the performance until we get to around 30 blocks. Adding more threads also increases performance until we get the expected drop 64 threads per block. We also get a slight drop after 32 threads since we passed the warp size and now have incomplete warps being scheduled. Figure 5 shows that the work stealing gives great performance and is not affected negatively by the increase in number of cores on the 8800GT. When we compared the task stealing with the other methods we used 64 threads and 32 blocks.

In Figure 4 (d) we see that the static method shows similar behavior as the task stealing. When increasing the number of threads used by the static method from 8 to 32 we get a steady increase in performance. Then we get the expected drops after 32 and 64, due to incomplete warps and less concurrent thread blocks. Increasing the number of threads further does not give any increase in speed as the synchronization overhead in the octree partitioning algorithm becomes dominant. The optimal number of threads for the static method is thus 32 and that is what we used when we compared it to the other methods in Figure 5.

As can be seen in Figure 4, adding more blocks than needed is not a problem since the only extra cost is an extra read of the finishing condition for each additional block.

11 Conclusions

We have compared four different load balancing methods, a blocking queue, a non-blocking queue, ABP task stealing and a static list, on the task of creating an octree partitioning of a set of particles.

We found that the blocking queue performed poorly and scaled badly when faced with more processing units, something which can be attributed to the inherent busy waiting. The non-blocking queue performed better but scaled poorly when the number of processing units got too high. Since the number of tasks increased quickly and the tree itself was relatively shallow the static queue performed well. The ABP task stealing method perform very well and outperformed the static method.

The experiments showed that synchronization can be very expensive and that new methods that take more advantage of the graphics processors features and capabilities might be required. They also showed that lock-free methods achieves better performance than blocking and that they can be made to scale with increased numbers of processing units.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[2] D. Cederman and P. Tsigas. A Practical Quicksort Algorithm for Graphics Processors. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA 2008), Lecture Notes in Computer Science Vol.: 5193*, pages 246–258. Springer-Verlag, 2008.

[3] D. Cederman and P. Tsigas. On Dynamic Load Balancing on Graphics Processors. In *Proceedings of the 11th Graphics Hardware (GH 2008)*, pages 57 – 64. ACM press, 2008.

[4] D. Cederman and P. Tsigas. GPU Quicksort Library. `www.cs.chalmers.se/~dcs/gpuqsortdcs.html`, December 2007.

[5] N. CUDA. www.nvidia.com/cuda.

[6] N. Govindaraju, N. Raghuvanshi, M. Henson, and D. Manocha. A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. Technical report, University of North Carolina-Chapel Hill, 2005.

[7] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, Aug. 2007.

[8] D. R. Helman, D. A. Bader, and J. JáJá. A Randomized Parallel Sorting Algorithm with an Experimental Study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, 1998.

[9] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(4):10–15, 1962.

[10] M. Matsumoto and T. Nishimura. Mersenne Twister: a 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[11] D. R. Musser. Introspective Sorting and Selection Algorithms. *Software - Practice and Experience*, 27(8):983–993, 1997.

[12] R. Sedgewick. Implementing Quicksort Programs. *Communications of the ACM*, 21(10):847–857, 1978.

[13] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan Primitives for GPU Computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 97–106, 2007.

[14] M. Shephard and M. Georges. Automatic three-dimensional mesh generation by the finite Octree technique. *International Journal for Numerical Methods in Engineering*, 32:709–749, 1991.

[15] R. C. Singleton. Algorithm 347: an Efficient Algorithm for Sorting with Minimal Storage. *Communications of the ACM*, 12(3):185–186, 1969.

[16] E. Sintorn and U. Assarsson. Fast Parallel GPU-Sorting Using a Hybrid Algorithm. In *Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[17] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, 2001.

Non-blocking Programming on Multi-core Graphics Processors (Extended Asbtract) *

Phuong Hoai Ha
University of Tromsø
Department of Computer Science
Faculty of Science, NO-9037 Tromsø, Norway
phuong@cs.uit.no

Philippas Tsigas
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg, Sweden
tsigas@chalmers.se

Otto J. Anshus
University of Tromsø
Department of Computer Science
Faculty of Science, NO-9037 Tromsø, Norway
otto@cs.uit.no

Abstract

This paper investigates the synchronization power of coalesced memory accesses, a family of memory access mechanisms introduced in recent large multicore architectures like the CUDA graphics processors. We first design three memory access models to capture the fundamental features of the new memory access mechanisms. Subsequently, we prove the exact synchronization power of these models in terms of their consensus numbers. These tight results show that the coalesced memory access mechanisms can facilitate strong synchronization between the threads of multicore processors, without the need of synchronization primitives other than reads and writes.

Moreover, based on the intrinsic features of recent GPU architectures, we construct strong synchronization objects like wait-free and t-resilient read-modify-write *objects for a general model of recent GPU architectures without strong hardware synchronization primitives like* test-and-set *and* compare-and-swap. *Accesses to the wait-free objects have time complexity $O(N)$, where N is the number of processes. Our result demonstrates that it is possible to construct wait-free synchronization mechanisms for GPUs without the need of strong synchronization primitives in hardware and that wait-free programming is possible for GPUs.*

*The results presented in this extended abstract appeared in the Proceedings of the 22nd International Symposium on Distributed Computing (DISC '08), ©Springer 2008 [12] and the Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08), ©IEEE 2008 [14].

1 Introduction

One of the fastest evolving multicore architectures is the graphics processor one. The computational power of graphics processors (GPUs) doubles every ten months, surpassing the Moore's Law for traditional microprocessors [21]. Unlike previous GPU architectures, which are single-instruction multiple-data (SIMD), recent GPU architectures (e.g. Compute Unified Device Architecture (CUDA) [2]) are single-program multiple-data (SPMD). The latter consists of multiple SIMD multiprocessors of which each, at the same time, can execute a different instruction. This extends the set of applications on GPUs, which are no longer restricted to follow the SIMD-programming model. Consequently, GPUs are emerging as powerful computational co-processors for general-purpose computations.

Along with their advances in computational power, GPUs memory access mechanisms have also evolved rapidly. Several new memory access mechanisms have been implemented in current commodity graphics/media processors like the Compute Unified Device Architecture (CUDA) [2] and Cell BE architecture [1]. For instance, in CUDA, single-word write instructions can write to words of different size and their size (in bytes) is no longer restricted to be a power of two [2]. Another advanced memory access mechanism implemented in CUDA is the coalesced global memory access mechanism. The simultaneous global memory accesses by each thread of a SIMD multiprocessor, during the execution of a single read/write instruction, are coalesced into a *single* aligned memory access if the simultaneous accesses follow the coalescence constraint [2]. The access coalescence takes place even if some of the threads do not actually access memory. It is well-known that memory access mechanisms, by devising how processing cores access the shared memory, directly influence the synchronization capabilities

of multicore processors. Therefore, it is crucial to investigate the synchronization power of the new memory access mechanisms.

Research on the synchronization power of memory access operations (or objects) in conventional architectures has received a great amount of attention in the literature. The synchronization power of memory access objects/mechanisms is conventionally determined by their consensus-solving ability, namely their consensus number [16]. The *consensus number* of an object type is either the maximum number of processes for which the consensus problem can be solved using only objects of this type and registers, or infinity if such a maximum does not exist.

1.1 The Synchronization Power of Coalesced Memory Accesses

We first investigate the consensus number of the new memory access mechanisms implemented in current graphics processor architectures. We design three new memory access models to capture the fundamental features of the new memory access mechanisms. Then we prove the exact synchronization power of these models in terms of their respective consensus number. These tight results show that the new memory access mechanisms can facilitate strong synchronization between the threads of multicore processors, without the need of synchronization primitives other than reads and writes [12].

Particularly, we design a new memory access model, the *svword* model where *svword* stands for the *s*ize-*v*arying *word* access, the first of the two aforementioned advanced memory access mechanisms implemented in CUDA. Unlike single-word assignments in conventional processor architectures, the new single-word assignments can write to words of size b (in bytes), where b can vary from 1 to an upper bound B and b is no longer restricted to be a power of 2 (e.g. type *float3* in [2]). By carefully choosing b for the single-word assignments, we can *partly* overlap the bytes written by two assignments, namely each of the two assignments has some byte(s) that is not overwritten by the other overlapping assignment Note that words of different size must be aligned from the address base of the memory. This memory alignment constraint prevents single-word assignments in conventional architectures from partly overlapping each other since the word-size is restricted to be a power of two. On the other hand, since the new single-word assignment can write to a subset of bytes of a *big* word (e.g. up to 16 bytes) and leave the other bytes of the word intact, the size of values to be written becomes a significant factor. The assignment can atomically write B values of size 1 (instead of just one value of size B) to B consecutive memory locations. The observation has motivated us to develop the *svword* model.

Inspired by the coalesced memory accesses, the second of the aforementioned advanced memory access mechanisms, we design two other models, the *aiword* and *asvword* models, to capture the fundamental features of the mechanism. The mechanism coalesces simultaneous read/write instructions by each thread of a SIMD multiprocessor into a *single* aligned memory access even if some of the threads do not actually access memory [2]. This allows each SIMD multiprocessor (or process) to atomically write to an arbitrary subset of the aligned memory units that can be written by a single coalesced memory access. We generally model this mechanism as an *a*ligned-*i*nconsecutive-*word* access, *aiword*, in which the memory is aligned to A-unit words and a single-word assignment can write to an arbitrary non-empty subset of the A units of a word. Note that the single-*aiword* assignment is not the atomic m-register assignment [16] due to the memory alignment restriction[1]. Our third model, *asvword*, is an extension of the second model *aiword* in which *aiword*'s A memory units are now replaced by A *svword*s of the same size b. This model is inspired by the fact that the read/write instructions of different coalesced global memory accesses can access words of different size [2].

1.2 Universal Synchronization Objects for GPUs

Subsequently, we design a set of universal synchronization objects capable of empowering the programmer with the necessary and sufficient tools for wait-free programming on graphics processors [14]. Based on the intrinsic features of recent GPU architectures, we first generalize the architectures to an abstract model of a chip with multiple SIMD cores sharing a memory. Each core can process many threads (in a SIMD manner) in one clock cycle. Each thread of a core accesses the shared memory using (atomic) read/write operations. Then, we construct wait-free and t-resilient synchronization objects [5, 16] for this model. The wait-free and t-resilient objects can be deployed as building blocks in parallel programming to help parallel applications tolerate crash failures and gain performance [19, 20, 25, 26].

We observe that due to SIMD architecture each SIMD core can read/write many memory locations in one atomic step. Using M-register read/write operations we construct a wait-free (long-lived) read-modify-write (RMW) objects in the case the number N of cores is not greater than $(2M-2)$. In the case $N > (2M-2)$, we construct $(2M-3)$-resilient RMW objects using only the M-register operations and read/write registers. It has been proved that $(2M-3)$ is the maximum number of crash failures that a system with M-register assignments and read/write registers can tolerate while ensuring consensus for correct processes[2] [8, 16]. Therefore, from a fault-tolerant point of view, these wait-free/resilient objects are the best we can achieve. To the best of our knowledge, research on constructing wait-free and $(2M-3)$-resilient long-lived RMW objects using only M-register read/write operations and read/write registers has not been reported previously.

[1] In this paper, we use term "single" in *single-*word assignment* when we want to emphasize that the assignment is not the *multiple* assignment [16].

[2] *Correct* processes are processes that do not crash in the execution.

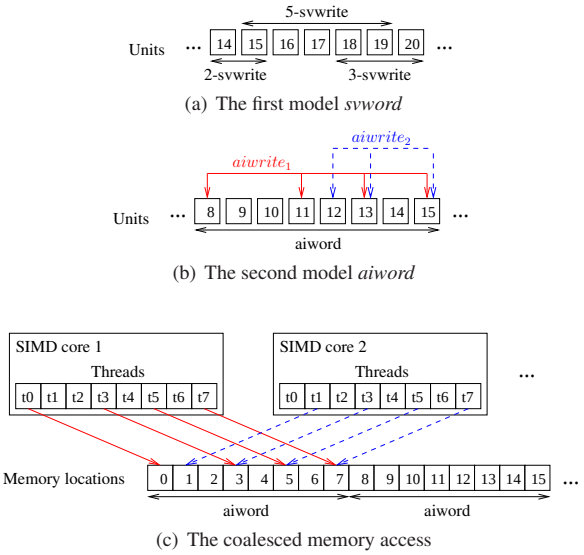
2 Coalesced Memory Accesses

2.1 Models

Before describing the details of each of the three new memory access models, we present the common properties of all these three models. The shared memory in the three new models is sequentially consistent [3, 18], which is weaker than the linearizable one [4] assumed in most of the previous research on the synchronization power of the conventional memory access models [16]. Processes are asynchronous. The new models use the conventional 1-dimensional memory address space. In these models, one memory *unit* is a *minimum* number of consecutive bytes/bits which a basic read/write operation can atomically read from/write to (without overwriting other unintended bytes/bits). These memory models address individual memory units. Memory is organized so that a group of n consecutive memory units called *word* can be stored or retrieved in a *single* basic write or read operation, respectively, and n is called *word size*. Words of size n must always start at addresses that are multiples of n, which is called *alignment restriction* as defined in the conventional computer architecture.

The first model is a *s*ize-*v*arying-*word* access model (*svword*) in which a single read/write operation can atomically read from/write to a word consisting of b *consecutive* memory units, where b can be any integer between 1 and an upper bound B and is called *svword size*. The upper bound B is the maximum number of *consecutive* units which a basic read/write operation can atomically read from/write to. *Sv-word*s of size b must always start at addresses that are multiples of b due to the memory alignment restriction. We denote *b-svword* to be an *svword* consisting of b units, *b-svwrite* to be a *b-svword* assignment and *b-svread* to be a *b-svword* read operation. Reading a unit U is denoted by 1-*svread*(U) or just by U for short. This model is inspired by the CUDA graphics processor architecture in which basic read/write operations can atomically read from/write to words of different size (cf. types *float1, float2, float3* and *float4* in [2], Section 4.3.1.1). Figure 1(a) illustrates how 2-*svwrite*, 3-*svwrite* and 5-*svwrite* can partly overlap their units with addresses from 14 to 20, with respect to the memory alignment restriction.

The second model is an *a*ligned-*i*nconsecutive-*word* access model (*aiword*) in which the memory is aligned to A-unit words and a single read/write operation can atomically read from/write to an arbitrary non-empty subset of the A units of a word, where A is a constant. *Aiword*s must always start at addresses that are multiples of A due to the memory alignment restriction. We denote A-*aiword* to be an *aiword* consisting of A units, A-*aiwrite* to be an A-*aiword* assignment and A-*airead* to be an A-*aiword* read operation. Reading only one unit U (using *airead*) is denoted by U for short. In the *aiword* model, an *aiwrite* operation executed by a process cannot *atomically* write to units located in different *aiword*s due to the memory alignment restriction.



(a) The first model *svword*

(b) The second model *aiword*

(c) The coalesced memory access

Figure 1. Illustrations for the first model, *s*ize-*v*arying-*word* access (*svword*), the second model, *a*ligned-*i*nconsecutive-*word* access (*aiword*) and the coalesced memory access.

Figure 1(b) illustrates the *aiword* model with $A = 8$ in which the *aiword* consists of eight consecutive units with addresses from 8 to 15. Unlike in the *svword* model, the assignment in the *aiword* model can atomically write to *inconsecutive* units of the eight units: $aiwrite_1$ atomically writes to four units $8, 11, 13$ and 15; $aiwrite_2$ writes to three units $12, 13$ and 15.

This model is inspired by the coalesced global memory accesses in the CUDA architecture [2]. The CUDA architecture can be generalized to an abstract model of a MIMD[3] chip with multiple SIMD cores sharing memory. Each core can process A threads simultaneously in a SIMD manner, but different cores can simultaneously execute different instructions. The instance of a program that is being sequentially executed by one SIMD core is called *process*. Namely, each process consists of A parallel threads that are running in SIMD manner. The process accesses the shared memory using the CUDA memory access models. In CUDA, the simultaneous global memory accesses by each thread of a SIMD core during the execution of a single read/write instruction can be coalesced into a *single* aligned memory access. The coalescence happens even if some of the threads do not actually access memory (cf. [2], Figure 5-1). This allows a SIMD core (or a process consisting of A parallel threads running in a SIMD manner) to atomically access multiple memory locations that are not at consecutive addresses.

Figure 1(c) illustrates the coalesced memory access, where $A = 8$. The left SIMD core can write atomically to four memory locations 0, 3, 5 and 7 by letting only four of

[3] MIMD: Multiple-Instruction-Multiple-Data

its eight threads, $t0, t3, t5$ and $t7$, simultaneously execute a write operation (i.e. divergent threads). The right SIMD core can write atomically to its own memory location 1 and shared memory locations 3, 5 and 7 by letting only four threads $t1, t3, t5$ and $t7$ simultaneously execute a write operation. Note that the CUDA architecture allows threads from different SIMD cores to communicate through the global shared memory [7].

The third model is a coalesced memory access model (*asvword*), an extension of the second model *aiword* in which *aiword*'s A units are now replaced by A *svword*s of the same size $b, b \in [1, B]$. Namely, the second model *aiword* is a special case of the third model *asvword* where $B = 1$. This model is inspired by the fact that in CUDA the read/write instructions of different coalesced global memory accesses can access words of different size. Let Axb-*asvword* be the *asvword* that is composed of A *svword*s of which each consists of b memory units. Axb-*asvword*s whose size is $A \cdot b$ must always start at addresses that are multiples of $A \cdot b$ due to the memory alignment restriction. We denote Axb-*asvwrite* to be an Axb-*asvword* assignment and Axb-*asvread* to be an Axb-*asvword* read operation. Reading only one unit U (using Ax1-*asvread*) is denoted by U for short. Due to the memory alignment restriction, an Axb-*asvwrite* operation cannot atomically write to b-*svword*s located in *different* Axb-*asvword*s. Since in reality A and B are a power of 2, in this model we assume that either $B = k \cdot A, k \in \mathbb{N}^*$ (in the case of $B \geq A$) or $A = k \cdot B, k \in \mathbb{N}^*$ (in the case of $B < A$). (At the moment, CUDA supports the *atomic* coalesced memory access to only words of size 4 and 8 bytes (i.e. only *svword*s consisting of 1 and 2 *units* in our definition), cf. Section 5.1.2.1 in [2]). For the sake of simplicity, we assume that $b \in \{1, B\}$ holds. A more general model with $b = 2^c, c = 0, 1, \cdots, \log_2 B$, can be established from this model. Since both Ax1-*asvword*s and AxB-*asvword*s are aligned from the address base of the memory space, any AxB-*asvword* can be aligned with B Ax1-*asvword*s as shown in Figure 2.

Figure 2 illustrates the *asvword* model in which each dash-dotted rectangle/square represents an *svword* and each red/solid rectangle represents an *asvword* composed of eight *svword*s (i.e. $A = 8$). The two rows show the memory alignment corresponding to the size b of *svword*s, where b is 1 or 2 (i.e. $B = 2$), on the same sixteen consecutive memory units with addresses from 0 to 15. An *asvwrite* operation can atomically write to some or all of the eight *svword*s of an *asvword*. Unlike the *aiwrite* assignment in the second model, which can atomically write to at most 8 units (or A units), the *asvwrite* assignment in the third model can atomically write to 16 units (or $A \cdot B$ units) using a single 8x2-*asvwrite* operation (i.e. write to the whole set of eight 2-*svword*s, cf. row $b = 2$). For an 8x1-*asvword* on row $b = 1$, there are two methods to update it atomically using the *asvwrite* operation: i) writing to the whole set of eight 1-*svword*s using a single 8x1-*asvwrite* (cf. SIMD core 1) or ii) writing to a subset consisting of four 2-*svword*s using a single 8x2-*asvwrite* (cf.

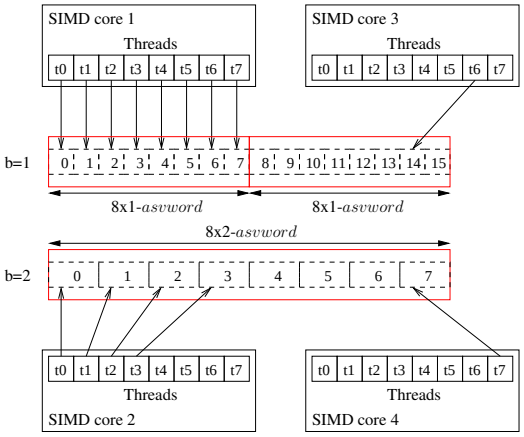


Figure 2. An illustration for the *asvword* model.

SIMD core 2). However, if only one of the eight units of an 8x1-*asvword* (e.g. unit 14) needs to be updated and the other units (e.g. unit 15) must remain untouched, the only possible method is to write to the unit using a single 8x1-*asvwrite* (cf. SIMD core 3). The other method, which writes to one 2-*svword* using a single 8x2-*asvwrite*, will have to overwrite another unit that is required to stay untouched (cf. SIMD core 4).

Terminology This section uses the conventional terminology from bivalency arguments [11, 16, 24]. The *configuration* of an algorithm at a moment in its execution consists of the state of every shared object and the internal state of every process. A configuration is *univalent* if all executions continuing from this configuration yield the same consensus value and *multivalent* otherwise. A configuration is *critical* if the next operation op_i by any process p_i will carry the algorithm from a *multivalent* to a *univalent* configuration. The operations op_i are called *critical operations*. The *critical value* of a process is the value that would get decided if that process takes the next step after the critical configuration.

2.2 Consensus number of the *svword* model

Before proving the consensus number of the single-*svword* assignment, we present the essential features of any wait-free consensus algorithm $\mathcal{ALG}$ for N processes using only single-**word* assignments and registers, where **word* can be *svword, aiword* or *asvword*. It has been proven that such an algorithm must have a critical configuration, C_0, and the next assignment op_i (i.e. the critical operation) by each process p_i must write to the same object $\mathcal{O}$ [16]. The object $\mathcal{O}$ consists of *memory units*.

Lemma 2.1. *The critical assignment op_i by each process p_i must atomically write to*

- *a "single-writer" unit (or 1W-unit for short) u_i written only by p_i and*

Algorithm 1 SVW_CONSENSUS(buf_i: proposal) invoked by process $p_i, i \in \{0, 1, 2\}$

$PROPOSAL[0, 1, 2]$: contains proposals of 3 processes. $PROPOSAL[i]$ is only written by process p_i but can be read by all processes.
WR_1 = set $\{u_0, u_1, u_2\}$ of $units$: initialized to $Init$ and used in the first phase. $WR_1[0]$ and $WR_1[2]$ are 1W-units written only by p_0 and p_1, respectively. $WR_1[1]$ is a 2W-unit written by both processes
WR_2 = set $\{v_0, \cdots, v_4\}$ of $units$: initialized to $Init$ and used in the second phase. $WR_2[0]$, $WR_2[2]$ and $WR_2[4]$ are 1W-units written only by p_0, p_2 and p_1, respectively. $WR_2[1]$ and $WR_2[3]$ are 2W-units written by pairs $\{p_0, p_2\}$ and $\{p_2, p_1\}$, respectively.
Input: process p_i's proposal value, buf_i.
Output: the value upon which all 3 processes (will) agree.
1V: $PROPOSAL[i] \leftarrow buf_i$; // Declare p_i's proposal
// **Phase 1**: Achieve an agreement between p_0 and p_1.
2V: **if** $i = 0$ or $i = 1$ **then**
3V: $first \leftarrow$ SVW_FIRSTAGREEMENT(i);
4V: **end if**
// **Phase II**: Achieve an agreement between all three processes.
5V: $winner \leftarrow$ SVW_SECONDAGREEMENT($i, first_{ref}$); // $first_{ref}$ is the reference to $first$
6V: **return** $PROPOSAL[winner]$

- *"two-writer" units (or 2W-units for short) $u_{i,j}$ written only by two processes p_i and p_j, where p_j's critical value is different from p_i's, $\forall j \neq i$.*

Proof. The proof is similar to the bivalency argument of Theorem 13 in [16]. □

In this section, we first present a wait-free consensus algorithm for 3 processes using only the single-*svword* assignment with $B \geq 5$ and registers. Then, we prove that we cannot construct any wait-free consensus algorithms for more than 3 processes using only the single-*svword* assignment and registers regardless of how large B is.

The new wait-free consensus algorithm SVW_CONSENSUS is presented in Algorithm 1. The main idea of the algorithm is to utilize the size-variation feature of the *svwrite* operation. Since *b-svwrite* can atomically write b values of size 1 unit (instead of just one value of size b units) to b consecutive memory units, keeping the size of values to be atomically written as small as 1 unit will maximize the number of processes for which *b-svwrite*, together with registers, can solve the consensus problem. Unlike the seminal wait-free consensus algorithm using the m-word assignment by Herlihy [16], which requires the word size to be large enough to accommodate a proposal value, the new algorithm stores proposal values in shared memory and uses only two bits (or one unit) to determine the preceding order between two processes. This allows a single-*svword* assignment to write atomically up to B (or $\frac{B}{2}$ if units are single bits) ordering-related values. The new algorithm utilizes process unique identifiers, which are an implicit assumption in Herlihy's consensus model [6].

The SVW_CONSENSUS algorithm has two phases. In the first phase, two processes p_0 and p_1 will achieve an agreement on their proposal values (cf. Algorithm 2). The agreed value, $PROPOSAL[first]$, is the proposal value of the *preceding process*, whose SVWRITE (lines 2SF and 4SF) precedes that of the other process (lines 6SF-11SF).

Due to the memory alignment restriction, in order to be able to allocate memory for the WR_1 variable (cf. Algorithm 1) on which p_0's and p_1's SVWRITEs can partly overlap, p_0's and p_1's SVWRITEs are chosen as 2-*svwrite* and 3-*svwrite*, respectively. The WR_1 variable is located in a memory region consisting of 4 consecutive units $\{u_0, u_1, u_2, u_3\}$ of which u_0 is at an address multiple of 2 and u_1 at an address multiple of 3. This memory allocation allows p_0 and p_1 to write atomically to the first two units $\{u_0, u_1\}$ and the last 3 units $\{u_1, u_2, u_3\}$, respectively (cf. Figure 3(a)). The WR_1 variable is the set $\{u_0, u_1, u_2\}$ (cf. the solid squares in Figure 3(a)), namely p_1 ignores u_3 (cf. line 4SF in Algorithm 2).

Algorithm 2 SVW_FIRSTAGREEMENT(i: bit) invoked by process $p_i, i \in \{0, 1\}$

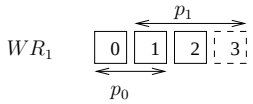
Output: the preceding process of $\{p_0, p_1\}$
1SF: **if** $i = 0$ **then**
2SF: SVWRITE($\{WR_1[0], WR_1[1]\}, \{Lower, Lower\}$); // atomically write to 2 units
3SF: **else**
4SF: SVWRITE($\{WR_1[1], WR_1[2]\}, \{Higher, Higher\}$); // $i = 1$
5SF: **end if**
6SF: **if** $WR_1[(\neg i) * 2] = \perp$ **then**
7SF: **return** i; // The other process hasn't written its value
8SF: **else if** ($WR_1[1] = Higher$ and $i = 0$) or ($WR[1] = Lower$ and $i = 1$) **then**
9SF: **return** i; // The other process comes later and overwrites p_i's value in $WR_1[1]$
10SF: **else**
11SF: **return** $(\neg i)$;
12SF: **end if**

Algorithm 3 SVW_SECONDAGREEMENT(i: index; $first_{ref}$: reference) invoked by process $p_i, i \in \{0, 1, 2\}$

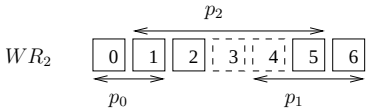
1SS: **if** $i = 0$ **then**
2SS: SVWRITE($\{WR_2[0], WR_2[1]\}, \{Lower, Lower\}$);
3SS: **else if** $i = 1$ **then**
4SS: SVWRITE($\{WR_2[3], WR_2[4]\}, \{Lower, Lower\}$);
5SS: **else**
6SS: SVWRITE($\{WR_2[1], WR_2[2], WR_2[3]\}, \{Higher, Higher, Higher\}$);
7SS: **end if**
8SS: **if** (($WR_2[0] \neq \perp$ or $WR_2[4] \neq \perp$) and $WR_2[2] = \perp$) or // The predicates are checked in the writing order.
($WR_2[0] \neq \perp$ and $WR_2[1] = Higher$) or
($WR_2[4] \neq \perp$ and $WR_2[3] = Higher$) **then**
9SS: **return** $first$; // p_2 is preceded by either p_0 or p_1. $first$ is obtained by dereferencing $first_{ref}$.
10SS: **else**
11SS: **return** 2;
12SS: **end if**

Subsequently, the agreed value will be used as the critical value of both p_0 and p_1 in the second phase in order to achieve an agreement with the other process p_2 (cf. Algorithm 3). Let p_{first} be the preceding process of p_0 and p_1 in the first phase. The second phase returns p_{first}'s proposal value if either p_0 or p_1 precedes p_2 (line 9SS) and returns p_2's proposal value otherwise.

Units written by processes' SVWRITE are illustrated in Figure 3(b). In order to be able to allocate memory for the WR_2 variable, process p_0's, p_1's and p_2's SVWRITEs are chosen as 2-*svwrite*, 3-*svwrite* and 5-*svwrite*, respectively. The WR_2 variable is located in a memory region consisting of 7 consecutive units $\{u_0, \cdots, u_6\}$ of which u_0 is at an address multiple of 2, u_4 at an address multiple of 3 and u_1 at an address multiple of 5. Since 2, 3 and 5 are prime numbers, we always can find such a memory region. For in-



(a) SVW_1stAgreement.



(b) SVW_2ndAgreement.

Figure 3. Illustrations for SVW_FIRSTAGREEMENT **and** SVW_SECONDAGREEMENT**.**

stance, if the memory address space starts from the unit with index 0, the memory region from unit 14 to unit 20 can be used for WR_2 (cf. Figure 1(a)). This memory allocation allows p_0, p_1 and p_2 to write atomically to the first two units $\{u_0, u_1\}$, the last three units $\{u_4, u_5, u_6\}$ and the five middle units $\{u_1, \cdots, u_5\}$, respectively. The WR_2 variable is the set $\{u_0, u_1, u_2, u_5, u_6\}$ (cf. the solid squares in Figure 3(b)).

Theorem 2.1. *The single-*svword *assignment has consensus number 3 when $B \geq 5$ and three is the upper bound of consensus numbers of single-*svword *assignments $\forall B \geq 2$.*

Proof. The full proof is in [13]. □

2.3 Consensus number of the *aiword* model

In this section, we prove that the single-*aiword* assignment (or *aiwrite* for short) has consensus number exactly $\lfloor \frac{A+1}{2} \rfloor$. First, we prove that the *aiwrite* operation has consensus number at least $\lfloor \frac{A+1}{2} \rfloor$. We prove this by presenting a wait-free consensus algorithm AIW_CONSENSUS for $N = \lfloor \frac{A+1}{2} \rfloor$ processes (cf. Algorithm 4) using only the *aiwrite* operation and registers. Subsequently, we prove that there is no wait-free consensus algorithm for $N+1$ processes using only the *aiwrite* operation and registers.

The main idea of the AIW_CONSENSUS algorithm is to gradually extend the set S of processes agreeing on the same value by one at a time. This is to minimize the number of 1W- and 2W-units that must be written atomically by the *aiword* operation. The algorithm consists of N rounds and a process $p_i, i \in [1, N]$, participates from round r_i to round r_N. A process p_i leaves a round $r_j, j \geq i$, and enters the next round r_{j+1} when it reads the value upon which all processes in the round r_j (will) agree. A round r_j starts with the first process that enters the round, and ends when all j processes $p_i, 1 \leq i \leq j$, have left the round. At the end of a round r_j, the set S consists of j processes $p_i, 1 \leq i \leq j$.

Lemma 2.2. *All correct processes*[4] *p_i agree on the same value in round r_j, where $1 \leq i \leq j \leq N$.*

[4] A *correct* process is a process that does not crash.

Algorithm 4 AIW_CONSENSUS(buf_i: proposal) invoked by process $p_i, i \in [1, N]$

$A^r[i]$: p_i's agreed value in round r;
$U^r_{i,j}$: the 2W-unit written only by processes p_i and p_j in round r. U^r_i: the 1W-unit written only by process p_i in round r;
Input: process p_i's proposal value, buf_i.
Output: the value upon which all N processes (will) agree.
// p_i starts from round i
1I: $A^i[i] \leftarrow buf_i$; // Initialized p_i's agreed value for round i
2I: AIWRITE($\{U^i_i, U^i_{i,1}, \cdots, U^i_{i,i-1}\}$, $\{Higher, Higher, \cdots, Higher\}$)
// Atomic assignment
3I: **for** $k = 1$ to $(i-1)$ **do**
4I: **if** $U^i_k \neq \perp$ and $U^i_{i,k} = Higher$ **then**
5I: $A^i[i] \leftarrow A^i[k]$; // Update p_i's agreed value to the set S's agreed value
6I: **break**;
7I: **end if**
8I: **end for**
// Participate rounds from $(i+1)$ to N
9I: **for** $j = i+1$ to N **do**
10I: $A^j[i] \leftarrow A^{j-1}[i]$; // Initialized p_i's agreed value for round j
11I: AIWRITE($\{U^j_i, U^j_{j,i}\}$, $\{Lower, Lower\}$); // Atomic assignment
12I: **if** $U^j_j \neq \perp$ and $U^j_{j,i} = Lower$ **then**
13I: $WinnerIsJ \leftarrow$ **true**; // Check if p_j precedes $p_k, \forall k < j$.
14I: **for** $k = 1$ to $j-1$ **do**
15I: **if** $U^j_k \neq \perp$ and $U^j_{j,k} = Higher$ **then**
16I: $WinnerIsJ \leftarrow$ **false**; // p_k precedes p_j;
17I: **break**;
18I: **end if**
19I: **end for**
20I: **if** $WinnerIsJ =$ **true then**
21I: $A^j[i] \leftarrow A^j[j]$; // p_j precedes $p_k, \forall k < j, \Rightarrow p_j$'s value is the agreed value in round j.
22I: **end if**
23I: **end if**
24I: **end for**
25I: **return** $A^N[i]$;

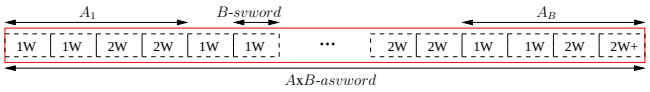


Figure 4. An illustration for grouping units in the *asvword* model.

With the assumption that AIWRITE can atomically write to p_j's units at line 2I and p_i's units at line 11I, it follows directly from Lemma 2.2 that all the N processes will achieve an agreement in round r_N.

Lemma 2.3. *The* AIW_CONSENSUS *algorithm is wait-free and can solve the consensus problem for $N = \lfloor \frac{A+1}{2} \rfloor$ processes.*

Theorem 2.2. *The single-*aiword *assignment has consensus number exactly $\lfloor \frac{A+1}{2} \rfloor$.*

Proof. The full proof is in [13]. □

2.4 Consensus number of the *asvword* model

The intuition behind the higher consensus number of the *asvword* model compared with the *aiword* model (cf. Equation (1)) is that process p_N in Algorithm 4 can atomically write to $A \cdot B$ units using AxB-*asvwrite* instead of only A units using A-*aiwrite*. To prevent p_N from overwriting unintended units (as illustrated by SIMD core 4 in Figure 2),

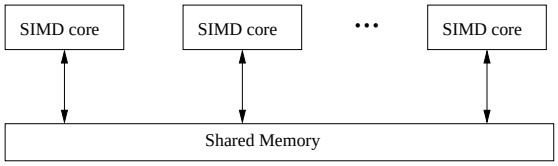


Figure 5. The abstract model of a chip with multiple SIMD-cores

each B-*svword* located in $A_l, 1 \leq l \leq B$, contains either 1W-units or 2W-units but not both as illustrated in Figure 4, where B-*svword*s labeled "1W" contain only 1W-units and B-*svword*s labeled "2W" contain only 2W-units. This allows p_N to atomically write to only B-*svword*s with 2W-units $U^N_{N,i}$ (and keep 1W-unit $U^N_i, i \neq N$, untouched) using AxB-*asvwrite*. For each process $p_i, i \neq N$, its 1W-unit U^N_i and 2W-unit $U^N_{N,i}$ are located in two B-*svword*s labeled "1W" and "2W", respectively, that belong to the same A_l. This allows p_i to atomically write to only its two units using Ax1-*asvwrite*. A complete proof of the exact consensus number can be found in the full version of this paper [13].

$$N = \begin{cases} \frac{AB}{2}, & \text{if } A = 2tB, t \in \mathbb{N}^* \text{ (positive integers)} \\ \frac{(A-B)B}{2} + 1, & \text{if } A = (2t+1)B, t \in \mathbb{N}^* \\ \lfloor \frac{A+1}{2} \rfloor, & \text{if } B = tA, t \in \mathbb{N}^* \end{cases} \quad (1)$$

3 Universal Synchronization Objects

3.1 The Model

Inspired by emerging media/graphics processing unit architectures like CUDA [2] and Cell BE [23], the abstract system model we consider in this paper is illustrated in Fig. 5. The model consists of N SIMD-cores sharing a shared memory and each core can process M threads (in a SIMD manner) in one clock cycle. For instance, the GeForce 8800GTX graphics processor, which is the flagship of the CUDA architecture family, has 16 SIMD-cores/SIMD-multiprocessors, each of which processes up to 16 concurrent threads in one clock cycle.

Since powerful media/graphics processing units with many cores (e.g. NVIDIA Tesla series with up to 64 cores and GeForce 8800 series with 16 cores) do not support strong synchronization primitives like *test-and-set* and *compare-and-swap* [2], we make no assumption on the existence of such strong synchronization primitives in this model. In this model, each of the M threads of one SIMD core can read/write one memory location in one atomic step. Due to SIMD architecture, each SIMD core can read/write M different memory locations in one atomic step or, in order words, each SIMD core can execute M_READ and M_ASSIGNMENT (atomic) operations.

Different cores can concurrently execute different user programs and a process, which sequentially executes instructions of a program on one core, can crash due to the program errors. The failure category considered in this model is the crash failure: a failed process cannot take another step in the execution. This model supports the strongly t-resilient formulation in which the access procedure at some port[5] of an object is infinite only if the access procedures in more than t other ports of the object are finite, nonempty and incomplete in the object execution [8].

Terminology Synchronization objects are conventionally classified by *consensus number*, the maximum number of processes for which the object can solve a consensus problem [16]. An n-*consensus object* allows n processes to propose their values and subsequently returns only one of these values to *all* the n processes. A *short-lived* (resp. *long-lived*) consensus object is a consensus object in which the object variables are used once (resp. many times) during the object life-time. An object implementation is *wait-free* if any process can complete any operation on the object in a finite number of steps regardless of the execution speeds of other processes [16, 17, 22]. An object implementation is *t-resilient* if non-faulty processes can complete their operations as long as no more than t processes fail [9, 11].

3.2 Wait-free Long-lived Consensus Objects Using M-assignment for $N = 2M-2$

In this section, we consider the following consensus problem. Each process is associated with a round number before participating in a consensus protocol. The round number must satisfy Requirement 3.1. The problem is to construct a long-lived object that guarantees consensus among processes with the same consensus number (or processes within the same round) using M_ASSIGNMENT operation. Since i) the adversary can arrange all N processes to be in the same round and ii) the M_ASSIGNMENT operation has consensus number $(2M-2)$, we cannot construct any wait-free objects that guarantee consensus for more than $(2M-2)$ processes using only the operation and read/write registers [16], or $N \leq (2M-2)$ must hold. The constructed wait-free long-lived consensus object will be used as a building block to construct wait-free read-modify-write objects in Section 3.3.

Requirement 3.1. *The requirements for processes' round number:*

- *a process' round number must be increasing and be updated only by this process,*
- *processes get a round number r only if the round $(r-1)$ has finished, and*
- *processes declare their current round number in shared variables before participating in a consensus protocol.*

[5] An object that allows N processes to access concurrently is considered having N ports.

Algorithm 5 LONGLIVEDCONSENSUS(buf_i: proposal) invoked by process p_i

$ROUND[1...N]$: contains current round numbers of N processes. $ROUND[i]$ is written by only process p_i and can be read by all N processes. $ROUND[i]$ must be set before p_i calls this LONGLIVEDCONSENSUS procedure.
$REG[\,][\,]$: 2-writer registers. $REG[i][j]$ can be written by processes p_i and p_j. For the sake of simplicity, we use a virtual array $2WR[1..M][1..M]$ that is mapped to REG of size $\frac{M(M-1)}{2}$ as follows

$$2WR[i][j] = \begin{cases} REG[i][j] & \text{if } i > j \\ REG[j][i] & \text{if } i < j \end{cases}$$

$1WR[1...M][0...1]$: 1-writer registers. $1WR[i]$ can be written by only process p_i. For implementation simplicity, $1WR[i]$ may contain $ROUND[i]$

Input: a unique proposal buf_i for p_i **and** p_i's round number $ROUND[i]$
Output: a proposal **or** $\perp$
1L: $gId \leftarrow \lfloor \frac{i}{M-1} \rfloor$ // Divide processes into 2 groups of size $(M-1)$ with group ID $gId \in \{0, 1\}$
// **Phase I:**Find an agreement in p_i's group with indices $\{gId(M-1)+1, \cdots, gId(M-1)+M-1\}$
2L: $first \leftarrow$ FIRSTAGREEMENT(buf_i, gId) // $first$ is the proposal of the earliest process of group gId in p_i's round
3L: **if** $first = \perp$ **then**
4L: **return** $\perp$ // p_i's round had finished and a new round started
5L: **end if**
// **Phase II**: Find an agreement with the other group with indices $\{(\neg gId)(M-1)+1, \cdots, (\neg gId)(M-1)+M-1\}$
6L: $winner \leftarrow$ SECONDAGREEMENT$(first, gId)$
7L: **if** $winner = \perp$ **then**
8L: **return** $\perp$ // pId's round had finished and a new round started
9L: **end if**
10L: **return** $winner$

Algorithm 6 FIRSTAGREEMENT(buf_i: proposal; gId: bit) invoked by process p_i

Output: $\perp$ **or** the proposal of the earliest process in p_i's round
1F: M_ASSIGNMENT$(\{1WR[i][gId], 2WR[i][\alpha+1], \cdots, 2WR[i][\alpha+M-1]\}, \{buf_i, \cdots, buf_i\})$, where $\alpha = gId(M-1)$
2F: $first \leftarrow i$ // Initialize the winner $first$ of p_i's group to p_i
3F: **for** k in $\alpha+1, \cdots, \alpha+M-1$ **do**
4F: $\{first, ref\} \leftarrow$ ORDERING$(first, k, gId)$ // Find the earliest process $first$ of p_i's group in p_i's round
5F: **if** $first = \perp$ **then**
6F: **return** $\perp$ // pId's round had finished and a new round started
7F: **end if**
8F: **end for**
9F: **return** ref // $first$'s proposal in p_i's round

At this moment, round numbers are assumed to be unbounded for the sake of simplicity. Solutions to make the round numbers bounded are presented in Section 3.5.

In the rest of this section, we presents a wait-free long-lived consensus (LLC) object for $N = (2M-2)$ processes using M_ASSIGNMENT operations. The LLC object is developed from the short-lived consensus (SLC) object using M_ASSIGNMENT in [16]. The LLC object will be used to achieve an agreement among processes in the same round. Unlike the SLC object, variables in the LLC object that are used in the current round can be reused in the next rounds. The LLC object, moreover, must handle the case that some processes (e.g. slow processes) belonging to other rounds try to modify the shared data/variables that are being used in the current round.

The algorithm of the wait-free LLC object using M_ASSIGNMENT is presented in Algo. 5. Before a process p_i invokes the LONGLIVEDCONSENSUS procedure, p_i's round number must be declared in the shared variable $ROUND[i]$. The procedure returns i) $\perp$ if p_i's round had finished and a newer round started or ii) one of the proposal data proposed in p_i's round.

A process p_i proposes its data by passing its proposal data to the procedure. Like the SLC object in [16], when the proposal data is unique for each process and can be stored in a register, the LLC object can work directly on the proposal data. However, when the proposal data is either not unique for each process or larger than the register size, which makes M_ASSIGNMENT no longer able to atomically write M proposal data, our LLC object works on the references to (or addresses of) the proposal data with the condition that processes allocate their own memory to contain their proposal data. In this case, applications using the LLC object must ensure that processes, after achieving an agreed reference ref, read the correct proposal data matching ref. Even though processes get the same reference ref via the LLC object, the data to which the reference refers may change, making processes get different data.

Like the SLC algorithm [16], the LLC algorithm divides the group of $(2M-2)$ processes into two fixed equal subgroups of $(M-1)$ processes (line 1L). In the first phase, the invoking process p_i finds the proposal of the earliest process of its group in its current round (line 2L). Then in the second phase, p_i uses the agreement achieved among its group in the first phase as its proposal for finding an agreement with its opposite group in its round (line 6L).

Note that p_i's round number is unchanged when p_i is executing the LONGLIVEDCONSENSUS procedure. If p_i's round already finished, the procedure returns $\perp$ since p_i is not allowed to participate in a consensus protocol of a round to which it doesn't belong (lines 4L and 8L).

The FIRSTAGREEMENT procedure (cf. Algo. 6) simply scans all members of p_i's group to find the earliest process using the ORDERING procedure. The ORDERING procedure receives as input two processes and returns the preceding one together with its proposal in p_i's round. Since the preceding order is transitive, the variable $first$ after the for-loop is the first process of p_i's group in p_i's round.

The SECONDAGREEMENT procedure (cf. Algo. 7) is an innovative improvement of the abstract idea in the SLC algorithm [16]. The SLC algorithm suggests the idea of constructing a directed graph between two groups each of $(M-1)$ processes with property that there is an edge from P_l to P_k if P_l and P_k are in different groups and the formers assignment precedes the latter's (or the former precedes the latter for short). Constructing such a directed graph has time complexity $O(M^2)$ since each member of one group must be checked with $(M-1)$ members of the other group.

However, the SECONDAGREEMENT procedure finds an agreement with time complexity only $O(M)$. The idea is

Algorithm 7 SECONDAGREEMENT($first$: proposal; gId: bit) invoked by process p_i

1S: M_ASSIGNMENT($\{1WR[i][\neg gId], 2WR[i][\beta + 1], \cdots, 2WR[i][\beta + M - 1]\}, \{first, \cdots, first\}$), where $\beta = (\neg gId)(M-1)$
2S: $winner \leftarrow i$ // Initialize the winner $winner$ to p_i
3S: $w\_gId \leftarrow gId$ // Initialize the winner's group ID w_gId
4S: $pivot[w\_gId] \leftarrow i$ // Set pivots for both groups to check all members of each group in a round-robin manner
5S: $pivot[\neg w\_gId] \leftarrow \beta + 1$ // The smallest index in $winner$'s opposite group
6S: $next \leftarrow pivot[\neg w\_gId]$
7S: **repeat**
8S: $previous \leftarrow winner$
9S: $\{winner, ref\} \leftarrow$ ORDERING($winner, next, \neg w\_gId$)
10S: **if** $winner = \perp$ **then**
11S: **return** $\perp$ // pId's round had finished and a new round started
12S: **else if** $winner \neq previous$ **then**
13S: $w\_gId \leftarrow \neg w\_gId$ // $winner$ now belongs to the other group
14S: $next \leftarrow previous$
15S: **end if**
16S: $next \leftarrow$ the next member index in $next$'s group in a round-robin manner.
17S: **until** $next = pivot[\neg w\_gId]$ // All members of $winner$'s opposite group have been checked
18S: **return** ref // The reference to $winner$'s proposal in round $round_i$

that we can find a process p_w in a group G_0 that precedes all members of the other group G_1 without the need of such a directed graph. Such a process is called *source*. Since all members of G_1 are preceded by p_w, they cannot be sources. All sources must be members of p_w's group G_0, which suggest the same proposal, their agreement achieved in the first phase. Therefore, all processes in both groups will achieve an agreement, the agreement of p_w's group.

The SECONDAGREEMENT procedure utilizes the transitive property of the preceding order to achieve the better time complexity $O(M)$. Fig. 6 illustrates the procedure. Assume that process p_i belongs to group 0, which is marked as p_i^0 in the figure. The procedure sets a pivot index for each group (e.g. $pivot^0 = p_i^0$ and $pivot^1 = p_1^1$) and checks members of each group in a round-robin manner starting from the group's pivot (lines 4S and 5S). In the figure, p_i^0, which is the temporary winner (line 2S), consecutively checks the members of group 1: p_1^1, p_2^1 and p_3^1, and discovers that it precedes p_1^1 and p_2^1 but it is preceded by p_3^1. At this point, the temporary winner $winner$ is changed from p_i^0 to p_3^1 and p_3^1 starts to checks the members of group 0 starting from p_{i+1}^0 (lines 12S-14S). Then, p_3^1 discovers that it precedes p_{i+1}^0 but it is preceded by p_{i+2}^0. At this point, the temporary winner $winner$ is again changed from p_3^1 to p_{i+2}^0. p_{i+2}^0 continues to check the members of group 1 *starting from* p_4^1, *the index before which* p_i^0 *stopped, instead of starting from* $pivot^1 = p_1^1$ (lines 12S-14S). It is clear from the figure that p_{i+2}^0 precedes p_1^1 and p_2^1 (or $p_{i+2}^0 \rightsquigarrow p_1^1$ and $p_{i+2}^0 \rightsquigarrow p_2^1$ for short) since $p_{i+2}^0 \rightsquigarrow p_3^1 \rightsquigarrow p_i^0$ and p_i^0 precedes both p_1^1 and p_2^1. Therefore, as long as the temporary winner (e.g. p_{i+2}^0) checks the $pivot$ of its opposite group again, it can ensure that it precedes all the members of its opposite group (line 17S) and becomes the final winner. Therefore, the procedure needs to check at most $(2M-2)$ times, leading to the time complexity $O(M)$.

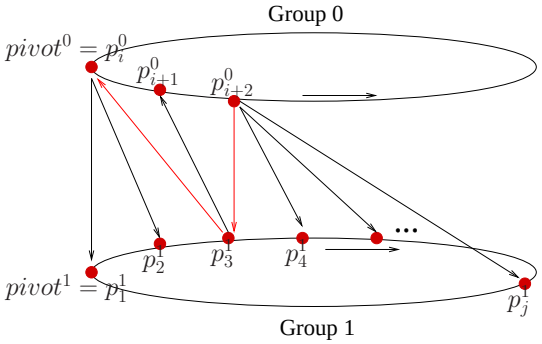


Figure 6. Illustration for the SECONDAGREEMENT **procedure, Algo. 7**

3.3 Wait-free Read-Modify-Write Objects for $N = 2M - 2$

In this section, we present a wait-free read-modify-write (RMW) object for $N = (2M-2)$ processes using M_ASSIGNMENT operations. Since the M_ASSIGNMENT operation has consensus number $(2M-2)$, we cannot construct any wait-free objects for more than $(2M-2)$ processes using only this operation and read/write registers [16]. The idea is to divide the execution of the RMW object into consecutive rounds. Processes belonging to the same round each suggests an order of these processes' functions to be executed on the object in that round, and then invokes the LONGLIVEDCONSENSUS procedure in Section 3.2 to achieve an agreement among these processes. Since each process executes one function on the RMW object at a time, functions are ordered according to both the round in which their matching processes participate and the agreed order among processes in the same round.

3.4 $(2M-3)$-Resilient Read-Modify-Write Objects for Arbitrary N

In this section, we present a $(2M-3)$-resilient object for an arbitrary number N of processes using M_ASSIGNMENT operations. Since the operation has consensus number $(2M-2)$, we cannot construct any objects that tolerate more than $(2M-3)$ faulty processes using only the M_ASSIGNMENT operation and read/write registers [8].

Let $D = (2M-2)$ and, without loss of generality, assume that $N = D^{\mathcal{K}}$, where $\mathcal{K}$ is an integer. The idea is to construct a balanced tree with degree of D. Processes start from the leaves at level $\mathcal{K}$ and climb up to the first level of the tree, the level just below the root. When visiting a node at level $i, 2 \leq i \leq \mathcal{K}$, a process p_i calls the wait-free LONGLIVEDCONSENSUS procedure (cf. Section 3.2) for its D sibling processes/nodes to find an agreement on which process will be their representative that will climb up to the higher level.

The representative process of p_i's D siblings at level l will participate in the wait-free LONGLIVEDCONSENSUS proce-

dure with its D siblings at level $(l+1)$ and so on until the representative reaches level 1 of the tree at which there are exact D nodes. At this level, the D processes/nodes invoke the wait-free RMW procedure for D processes (cf. Section 3.3).

Processes that are not chosen to be the representative stop climbing the tree and repeatedly check the final result until their function is executed. After that they return with the corresponding response.

3.5 Bounded round numbers

Active processes p_i that are participating in the most recent instance of the long-lived protocol need a mechanism to distinguish them from slow/sleepy processes. The bounded version of the long-lived protocol can be obtained by replacing the unbounded round number with the (bounded) *leadership graph* suggested in [15]. In the graph, an incoming process p_i invokes the ADVANCE operation to become one of the leaders of the graph. Processes that are current leaders belong to the most recent round whereas processes that are no longer leaders are slow processes. Therefore, the leadership graph can help distinguish active processes from slow processes, satisfying the requirement of the long-lived protocol. Another approach to bound the round number is to use the transforming technique presented in [10]. The technique can transform any unbounded algorithm based on an asynchronous rounds structure into a bounded algorithm in a way that preserves correctness and running time.

References

[1] *Cell Broadband Engine Architecture, version 1.01*. IBM, Sony and Toshiba Corporations, 2006.

[2] *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, version 1.1*. NVIDIA Corporation, 2007.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley and Sons, Inc., 2004.

[5] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 91–100, 1993.

[6] H. Buhrman, A. Panconesi, R. Silvestri, and P. Vitanyi. On the importance of having an identity or, is consensus really universal? *Distrib. Comput.*, 18(3):167–176, 2006.

[7] I. Castano and P. Micikevicius. Personal communication. *NVIDIA*, 2008.

[8] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. *SIAM Journal on Computing*, 34(2):333–357, 2005.

[9] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[10] C. Dwork and M. Herlihy. Bounded round number. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 53–64, 1993.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[12] P. H. Ha, P. Tsigas, and O. J. Anshus. The synchronization power of coalesced memory accesses. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 320–334, 2008.

[13] P. H. Ha, P. Tsigas, and O. J. Anshus. The synchronization power of coalesced memory accesses. *Technical report CS:2008-68, University of Tromsø, Norway*, 2008.

[14] P. H. Ha, P. Tsigas, and O. J. Anshus. Wait-free programming for general purpose computations on graphics processors. In *Proc. of the IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–12, 2008.

[15] M. Herlihy. Randomized wait-free concurrent objects (extended abstract). In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 11–21, 1991.

[16] M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, Jan. 1991.

[17] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[19] S. S. Lumetta and D. E. Culler. Managing concurrent access for shared memory active messages. In *Proc. of the Intl. Parallel Processing Symp. (IPPS)*, page 272, 1998.

[20] M. M. Michael and M. L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proc. of the IEEE Intl. Parallel Processing Symp. (IPPS*, pages 267–273, 1997.

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[22] G. L. Peterson. Concurrent reading while writing. *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[23] D. Pham and et.al. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–185, 2005.

[24] E. Ruppert. Determining consensus numbers. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, pages 93–99, 1997.

[25] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 320–321, 2001.

[26] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02)*, pages 55–67, July 2002.

Paper session 2: Language and compilation techniques

OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems

Shuvra S. Bhattacharyya
Dept. of ECE and UMIACS
University of Maryland, College Park, MD 20742
USA

Gordon Brebner, Jörn W. Janneck
Xilinx Research Labs
San Jose, CA 95123
USA

Johan Eker, Carl von Platen
Ericsson Research
Mobile Platforms
SE-221 83, Lund
Sweden

Marco Mattavelli
Microelectronic Systems Lab
EPFL
CH-1015 Lausanne
Switzerland

Mickaël Raulet
IETR/INSA Rennes
F-35043, Rennes
France

Abstract

This paper presents the OpenDF framework and recalls that dataflow programming was once invented to address the problem of parallel computing. We discuss the problems with an imperative style, von Neumann programs, and present what we believe are the advantages of using a dataflow programming model. The CAL actor language is briefly presented and its role in the ISO/MPEG standard is discussed. The Dataflow Interchange Format (DIF) and related tools can be used for analysis of actors and networks, demonstrating the advantages of a dataflow approach. Finally, an overview of a case study implementing an MPEG-4 decoder is given.

1 Introduction

Time after time, the uniprocessor system has managed to survive in spite of rumors of its imminent death. Over the last three decades hardware engineers have been able to achieve performance gains by increasing clock speed, and introducing cache memories and instruction level parallelism. However, current developments in the hardware industry clearly shows that this trend is over. The frequency in no longer increasing, but instead the number of cores on each CPU is. Software development for uniprocessor systems is completely dominated by imperative style programming models, such as C or Java. And while they provide a suitable abstraction level for uniprocessor systems, they fail to do the same in a multicore setting. In a time when new hardware meant higher clock frequencies, old programs almost always ran faster on more modern equipment. However, this is not true when programs written for single core system execute on multicore. And the bad news is that there is no easy way of modifying them. Tools such as OpenMP will help the transition, but likely fail to utilize the full potential of multicore systems.

Over the years considerable attention has been put to the data flow modeling, which is a programming paradigm proposed in the late 60s, as a means to address parallel programming. It is well researched area with a number of interesting results pertaining to parallel computing. Many modern forms of computation are very well suited for data flow description and implementation, examples are complex media coding [1], network processing [2], imaging and digital signal processing [3], as well as embedded control [4]. Together with the move toward parallelism, this represents a huge opportunity for data flow programming.

2 Why C etc. Fail

Before diving into dataflow matters, we will give a brief motivation why a paradigm shift is necessary. The control over low-level detail, which is considered a merit of C, tends to over-specify programs: not only the algorithms themselves are specified, but also how inherently parallel computations are sequenced, how inputs and outputs are passed between the algorithms and, at a higher level, how computations are mapped to threads, processors and application-specific hardware. It is not always possible to recover the original knowledge about the program by means of analysis and the opportunities for restructuring transformations are limited.

Code generation is constrained by the requirement of preserving the semantic effect of the original program. What constitutes the semantic effect of a program depends on the source language, but loosely speaking some observable properties of the program's execution are required to be invariant. Program analysis is employed to identify the set of admissible transformations; a code generator is required to be conservative in the sense that it can only perform a particular transformation when the analysis results can be used to prove that the effect of the program is preserved. *Dependence analysis* is one of the most challenging tasks of high-quality code generation (for instance see [5]). It determines a set of constraints on the order, in which the computations of a program may be performed. Efficient utilization of modern processor architectures heavily depends on dependence analysis, for instance:

- To determine efficient mappings of a program onto multiple processor cores (*parallelization*),
- to utilize so called SIMD or "multimedia" instructions that operate on multiple scalar values simultaneously (*vectorization*), and
- to utilize multiple functional units and avoid pipeline stalls (*instruction scheduling*).

Determining (a conservative approximation of) the dependence relation of a C program involves *pointer analysis*. Since the general problem is undecideable, a trade-off will always have to be made between the precision of the analysis and its resource requirements [6].

3 Dataflow Networks

A dataflow program is defined as a directed graph, where the nodes represent computational units and the arcs represent the flow of data. The lucidness of dataflow graphs can be deceptive. To be able to reason about the effect of the computations performed, the dataflow graph has to be put in the context of a computation model, which defines the semantics of the communication between the nodes. There exists a variety of such models, which makes different trade-offs between expressiveness and analyzability. Of particular interest are Kahn process networks [7], and synchronous dataflow networks [8]. The latter is more constrained and allows for more compile-time analysis for calculation of static schedules with bounded memory, leading to synthesized code that is particularly efficient. More general forms of dataflow programs are usually scheduled dynamically, which induces a run-time overhead.

It has been shown that dataflow models offer a representation that can effectively support the tasks of parallelization [8] and vectorization [9]—thus providing a practical means of supporting multiprocessor systems and utilizing vector instructions.

3.1 Actors

The fundamental entity of this model is an *actor* [10], also called dataflow actor with firing. Dataflow graphs, called *networks*, are created by means of connecting the *input* and *output ports* of the actors. Ports are also provided by networks, which means that networks can nested in a hierarchical fashion. Data is produced and consumed as *tokens*, which could correspond to samples or have a more complex structure. This model has the following properties:

- **Strong encapsulation.** Every actor completely encapsulates its own state together with the code that operates on it. No two actors ever share state, which means that an actor cannot directly read or modify another actor's state variables. The only way actors can interact is through streams, directed connections they use to communicate data tokens.
- **Explicit concurrency.** A system of actors connected by streams is explicitly concurrent, since every single actor operates independently from other actors in the system, subject to dependencies established by the streams mediating their interactions.
- **Asynchrony, untimedness.** The description of the actors as well as their interaction does not contain specific real-time constraints (although, of course, implementations may).

4 The CAL Actor Language

CAL [11] is a domain-specific language that provides useful abstractions for dataflow programming with actors. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations, and work on mixed HW/SW implementations is under way. Below we will give a brief introduction to some key elements of the language.

4.1 Basic Constructs

The basic structure of a CAL actor is shown in the `Add` actor below, which has two input ports `t1` and `t2`, and one output port `s`, all of type `T`. The actor contains one *action* that consumes one token on each input ports, and produces one token on the output port. An action may *fire* if the availability of tokens on the input ports matches the *port patterns*, which in this example corresponds to one token on both ports `t1` and `t2`.

```
actor Add() T t1, T t2 ⇒ T s :
  action [a], [b] ⇒ [sum]
  do
     sum := a + b;
  end
end
```

An actor may have any number of actions. The untyped `Select` actor below reads and forwards a token from either port `A` or `B`, depending on the evaluation of guard conditions. Note that each of the actions have empty bodies.

```
actor Select () S, A, B ⇒ Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

An action may be labeled and it is possible to constrain the legal firing sequence by expressions over labels. In the `PingPongMerge` actor, see below, a finite state machine *schedule* is used to force the action sequence to alternate between the two actions `A` and `B`. The schedule statement introduces two states `s1` and `s2`.

```
actor PingPongMerge () Input1, Input2 ⇒ Output:
  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

The `Route` actor below forwards the token on the input port `A` to one of the three output ports. Upon instantiation it takes two parameters, the functions `P` and `Q`, which are used as predicates in the guard conditions. The selection of which action to fire is in this example not only determined by the availability of tokens and the guards conditions, by also depends on the `priority` statement.

```
actor Route (P, Q) A ⇒ X, Y, Z:

    toX: action [v] ⇒ X: [v]
         guard P(v) end

    toY: action [v] ⇒ Y: [v]
         guard Q(v) end

    toZ: action [v] ⇒ Z: [v] end

    priority
        toX > toY > toZ;
    end
end
```

For an in-depth description of the language, the reader is referred to the language report [11]. A large selection of example actors is available at the OpenDF repository, among them the MPEG-4 decoder discussed below.

4.2 Networks

A set of CAL actors are instantiated and connected to form a CAL application, i.e. a CAL network. Figure 1 shows a simple CAL network `Sum`, which consists of the previously defined `Add` actor and the delay actor shown below.

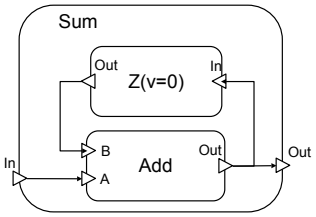


Figure 1. A simple CAL network.

```
actor Z (v) In ⇒ Out:

  A: action ⇒ [v] end
  B: action [x] ⇒ [x] end

  schedule fsm s0:
    s0 (A) --> s1;
    s1 (B) --> s1;
  end
end
```

The source that defined the network `Sum` is found below. Please, note that the network itself has input and output ports and that the instantiated entities may be either actors or other networks, which allows for a hierarchical design.

```
network Sum () In ⇒ Out:

entities
  add = Add();
  z = Z(v=0);

structure
  In --> add.A;
  z.Out --> add.B;

  add.Out --> z.In;

  add.Out -- > Out;
end
```

4.3 ISO-MPEG standardisation

The data-driven programming paradigm of CAL dataflow lends itself naturally to describing the processing of media streams that pervade the world of media coding. In addition, the strong encapsulation afforded by the actor model provides a solid foundation for the modular specification of media codecs.

MPEG has produced several video coding standards such as MPEG-1, MPEG-2, MPEG-4 Video, AVC and SVC. However, the past monolithic specification of such standards (usually in the form of C/C++ programs) lacks flexibility and does not allow to use the combination of coding algorithms from different standards enabling to achieve specific design or performance trade-offs and thus fill, case by case, the requirements of specific applications. Indeed, not all coding tools defined in a *profile@level* of a specific standard are required in all application scenarios. For a given

application, codecs are either not exploited at their full potential or require unnecessarily complex implementations. However, a decoder conformant to a standard has to support all of them and may results in a non-efficient implementation.

So as to overcome the limitations intrinsic of specifying codecs algorithms by using monolithic imperative code, CAL language has been chosen by the ISO/IEC standardization organization in the new MPEG standard called Reconfigurable Video Coding (RVC) (ISO/IEC 23001-4 and 23002-4). RVC is a framework allowing users to define a multitude of different codecs, by combining together actors (called coding tools in RVC) from the MPEG standard library written in CAL, that contains video technology from all existing MPEG video past standards (i.e. MPEG-2, MPEG- 4, etc.). The reader can refer to [12] for more information about RVC. CAL is used to provide the reference software for all coding tools of the entire library. The essential elements of the RVC framework include:

- the standard Video Tool Library (VTL) which contains video coding tools, also named Functional Units (FU). CAL is used to describe the algorithmic behaviour of the FUs that end to be video coding algorithmic components self contained and communicating with the external world only by means of input and output ports.
- a language called Functional unit Network Language (FNL), an XML dialect, used to specify a decoder configuration made up of FUs taken from the VTL and the connections between the FUs.
- a MPEG-21 Bitstream Syntax Description Language (BSDL) schema which describes the syntax of the bitstream that a RVC decoder has to decode. A BSDL to CAL translator is under development as part of the OpenDF effort.

In summary the components and processes that lead to the specification and implementation of a new MPEG RVC decoder are based on the CAL dataflow model of computation and are:

- a Decoder Description (DD) written in FNL describing the architecture of the decoder, in terms of FUs and their connections.
- an Abstract Decoder Model (ADM), a behavioral (CAL) model of the decoder composed of the syntax parser specified by the BSDL schema, FUs from the VTL and their connections.
- the final decoder implementation that is either generated by substituting any proprietary implementation, conformant in terms of I/O behavior, of the standard RVC FUs, or obtained directly from the ADM by generating SW and/or HW implementations by means of appropriate synthesis tools.

Thus, based on CAL dataflow formalism, designers can build video coding algorithm with a set of self-contained modular elements coming from the MPEG RVC standard library (VTL). However, the new CAL based specification formalism, not only provide the flexibility required by the process itself of specifying a standard video codec, but also yields a specification of such standard that is the appropriate starting point for the implementation of the codec on the new generations of multicore platforms. In fact the RVC ADM is nothing else that a CAL datatflow specification that implicitly expose all concurrency and parallelism intrinsic to the model, features that classical generic specifications based on imperative languages have not provided.

5 Tools

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses[1] project. Moses features a graphical network editor, and allows the user to monitor actors execution (actor state and token values). The project being no longer maintained, it has been superseded by the Open Dataflow environment (OpenDF[2] for short).

OpenDF is also a compilation framework. Today there exists a backend for generation of HDL(VHDL/Verilog) [13], and another backend for that generates C for integration with the SystemC tool chain [14]. A third backend targeting ARM11 and embedded C is under development [15] as part of the EU project ACTORS[3]. It is also possible to simulate CAL models in the Ptolemy II[4] environment.

5.1 Analysis Support

A related tool is the dataflow interchange format (DIF), which is a textual language for specifying mixed-grain dataflow representations of signal processing applications, and TDP[5] (the DIF package), which is a software tool for analyzing DIF specifications. A major emphasis in DIF and TDP is support for working with and integrating different kinds of specialized dataflow models of computation and their associated analysis techniques. Such functionality is useful, for example, as a follow-on step to the automated detection of specialized dataflow regions in CAL networks. Once such regions are detected, they can be annotated with corresponding DIF keywords — e.g., CSDF

[1] http://www.tik.ee.ethz.ch/ moses/
[2] http://opendf.sourceforge.net
[3] http://www.actors-project.eu
[4] http://ptolemy.eecs.berkely.edu
[5] http://www.ece.umd.edu/DSPCAD/dif

(cyclo-static dataflow) and SDF (synchronous dataflow) — and then scheduled and integrated with appropriate TDP-based analysis methods. Such a linkage between CAL and TDP is under active development as a joint effort between the CAL and DIF projects.

A particular area of emphasis in TDP is support for developing efficient coarse-grain dataflow scheduling techniques. For example, the generalized schedule tree representation in TDP provides an efficient format for storing, manipulating, and viewing schedules [16], and the functional DIF dataflow model provides for flexible prototyping of static, dynamic, and quasi-static scheduling techniques [3]. Libraries of static scheduling techniques and buffer management models for SDF graphs, as well as an SDF-to-C translator are also available in TDP [17]. The set of dataflow models that are currently recognized and supported explicitly in the DIF language and TDP include Boolean dataflow [18], enable-invoke dataflow [3], CSDF [19], homogeneous synchronous dataflow [8, 20], multidimensional synchronous dataflow [21], parameterized synchronous dataflow [22], and SDF [8]. These alternative dataflow models have useful trade-offs in terms of expressive power, and support for efficient static or quasi-static scheduling, as well as efficient buffer management. The set of models that is supported in TDP, as well as the library of associated analysis techniques are expanding with successive versions of the TDP software.

The initial focus in integrating TDP with CAL is to automatically-detect regions of CAL networks that conform to SDF semantics, and can leverage the significant body of SDF-oriented analysis techniques in TDP. In the longer term, we plan to target a range of different dataflow models in our automated "region detection" phase of the design flow. This appears significantly more challenging as most other models are more complex in structure compared to SDF; however, it can greatly increase the flexibility with which different kinds of specialized, streaming-oriented dataflow analysis techniques can be leveraged when synthesizing hardware and software from CAL networks.

6 Why dataflow might actually work

Scalable parallelism. In parallel programming, the number of things that are happening at the same time can scale in two ways: It can increase with the size of the problem or with the size of the program. Scaling a regular algorithm over larger amounts of data is a relatively well-understood problem, while building programs such that their parts execute concurrently without much interference is one of the key problems in scaling the von Neumann model. The explicit concurrency of the actor model provides a straightforward parallel composition mechanism that tends to lead to more parallelism as applications grow in size, and scheduling techniques permit scaling concurrent descriptions onto platforms with varying degrees of parallelism.

- **Modularity, reuse.** The ability to create new abstractions by building reusable entities is a key element in every programming language. For instance, object-oriented programming has made huge contributions to the construction of von Neumann programs, and the strong encapsulation of actors along with their hierarchical composability offers an analog for parallel programs.

- **Scheduling.** In contrast to procedural programming languages, where control flow is made explict, the actor model emphasizes explicit specification of concurrency.

- **Portability.** Rallying around the pivotal and unifying von Neumann abstraction has resulted in a long and very successful collaboration between processor architects, compiler writers, and programmers. Yet, for many highly concurrent programs, portability has remained an elusive goal, often due to their sensitivity to timing. The untimedness and asynchrony of stream-based programming offers a solution to this problem.

  The portability of stream-based programs is evidenced by the fact that programs of considerable complexity and size can be compiled to competitive hardware [13] as well as software [14], which suggests that stream-based programming might even be a solution to the old problem of flexibly co-synthesizing different mixes of hardware/software implementations from a single source.

- **Adaptivity.** The success of a stream programming model will in part depend on its ability to configure dynamically and to virtualize, i.e. to map to collections of computing resources too small for the entire program at once. The transactional execution of actors generates points of *quiescence*, the moments between transactions, when the actor is in a defined and known state that can be safely transferred across computing resources.

7 The MPEG-4 Case Study

One interesting usage of the collection of CAL actors, which constitutes the MPEG RVC tools library, is as a vehicle for video coding experiments. Since it provides a source of relevant application of realistic sizes and complexity, the tools library also enables experiments in dataflow programming, the associated development process and development tools.

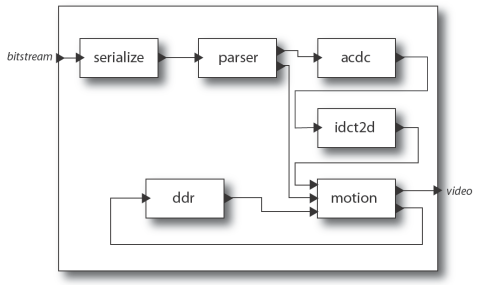


Figure 2. Top-level dataflow graph of the MPEG-4 decoder.

Some of the authors have performed a case study[13], in which the MPEG-4 Simple Profile decoder was specified in CAL and implemented on an FPGA using a CAL-to-RTL code generator. Figure 2 shows a top-level view of decoder. The main functional blocks include a bitstream parser, a reconstruction block, a 2D inverse cosine transform, a frame buffer and a motion compensator. These functional units are themselves hierarchical compositions of actor networks. The objective of the design was to support 30 frames of 1080p in the YUV420 format per second, which amounts to a production of 93.3 Mbyte of video output per second. The given target clock rate of 120 MHz implies 1.29 cycles of processing per output sample on average.

The results of the case study were encouraging in that the code generated from the CAL specification did not only outperformed the handwritten reference in VHDL, both in terms of throughput and silicon area, but also allowed for a significantly reduced development effort. Table 3 shows the comparison between CAL specification and the VHDL reference.

It should be emphasized that this counter-intuitive result cannot be attributed to the sophistication of the synthesis tool. On the contrary the tool does not perform a number of potential optimizations; particularly it does not consider optimizations involving more than one actor. Instead, the good results appear to be due to the development process. A notable difference was that the CAL specification went through significantly more design iterations than the VHDL reference —in spite of being performed in a quarter of the development time. Whereas a dominant part of the development of the VHDL reference was spent getting the system to work correctly, the effort of the CAL specification was focused on optimizing system performance to meet the design constraints.

The initial design cycle resulted in an implementation that was not only inferior to the VHDL reference, but one that also failed to meet the throughput and area constraints. Subsequent iterations explored several other points in the design space until arriving at a solution that satisfied the constraints. At least for the case study, the benefit of short design cycles seem to outweigh the inefficiencies that were induced by high-level synthesis and the reduced control over implementation details.

	Size slices, BRAM	**Speed** kMB/S	**Code size** kLOC	**Dev. time** MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12
Improv. factor	1.2	1.6	3.75	4

Figure 3. Hardware synthesis results for an MPEG-4 Simple Profile decoder. The numbers are compared with a reference hand written design in VHDL.

In particular, the asynchrony of the programming model and its realization in hardware allowed for convenient experiments with design ideas. Local changes, involving only one or a few actors, do not break the rest of the system in spite of a significantly modified temporal behavior. In contrast, any design methodology that relies on precise specification of timing —such as RTL, where designers specify behavior cycle-by-cycle— would have resulted in changes that propagate through the design.

Figure 3 shows the quality of result produced by the RTL synthesis engine for a real-world application, in this case an MPEG-4 Simple Profile video decoder. Note that the code generated from the high-level dataflow description actually outperforms the VHDL design in terms of both throughput and silicon area for a FPGA implementation.

8 Summary

We believe that the move towards parallelism in computing and the growth of application areas that lend themselves to dataflow modeling present a huge opportunity for a dataflow programming model that could supplant or at least complement von Neumann computing in many fields.

We have discussed some properties that comes with using a dataflow model, such as explicit parallelism and decoupling of scheduling and communication. The open source simulation and compilation framework OpenDF was presented together with the CAL language and the DIF/TDP analysis tools. Finally, the work on the MPEG-4 decoder verifies the potential of the dataflow approach.

References

[1] J. Thomas-Kerr, J. W. Janneck, M. Mattavelli, I. Burnett, and C. Ritz, “Reconfigurable Media Coding:

Self-describing multimedia bitstreams," in *Proceedings IEEE Workshop on Signal Processing Systems—SiPS 2007*, October 2007, pp. 319–324.

[2] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999.

[3] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[4] S. S. Bhattacharyya and W. S. Levine, "Optimization of signal processing software for control system implementation," in *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, Munich, Germany, October 2006, pp. 1562–1567, invited paper.

[5] H. Zima and B. Chapman, *Supercompilers for parallel and vector computers*. New York, NY, USA: ACM, 1991.

[6] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2001, pp. 54–61.

[7] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

[8] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[9] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Intl. Conf. on Application-Specific Array Processors*. Prentice Hall, IEEE Computer Society, 1993, pp. 285–296.

[10] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artif. Intell.*, vol. 8, no. 3, pp. 323–364, 1977.

[11] J. Eker and J. W. Janneck, "Cal language report," University of California at Berkeley, Tech. Rep. UCB/ERL M03/48, December 2003.

[12] C. Lucarz and J. J. Marco Mattavelli, Joseph Thomas-Kerr, "Reconfigurable media coding: A new specification model for multimedia coders," in *Proceedings of IEEE Workshop on Signal Processing Systems*, 2007, pp. 481–486.

[13] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008.

[14] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008.

[15] C. von Platen and J. Eker, "Efficient realization of a cal video decoder on a mobile terminal," in *Proceedings of IEEE Workshop on Signal Processing Systems*, 2008.

[16] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, "Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.

[17] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[18] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[19] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[20] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc., 2000.

[21] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, August 2002.

[22] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.

Optimized On-Chip Pipelining of Memory-Intensive Computations on the Cell BE

Christoph W. Kessler
Linköpings Universitet
Dept. of Computer and Inf. Science
58183 Linköping, Sweden
chrke@ida.liu.se

Jörg Keller
FernUniversität in Hagen
Dept. of Mathematics and Computer Science
58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

Abstract

Multiprocessors-on-chip, such as the Cell BE processor, regularly suffer from restricted bandwidth to off-chip main memory. We propose to reduce memory bandwidth requirements, and thus increase performance, by expressing our application as a task graph, by running dependent tasks concurrently and by pipelining results directly from task to task where possible, instead of buffering in off-chip memory. To maximize bandwidth savings and balance load simultaneously, we solve a mapping problem of tasks to SPEs on the Cell BE. We present three approaches: an integer linear programming formulation that allows to compute Pareto-optimal mappings for smaller task graphs, general heuristics, and a problem specific approximation algorithm. We validate the mappings for dataparallel computations and sorting.

1. Introduction

The new generation of multiprocessors-on-chip derives its raw power from parallelism, and explicit parallel programming with platform-specific tuning is needed to turn this power into performance. A prominent example is the Cell Broadband Engine with a PowerPC core and 8 parallel slave processors called SPEs (e.g. cf. [1]). Yet, many applications use the Cell BE like a dancehall architecture: the SPEs use their small on-chip local memories (256 KB for both code and data) as explicitly-managed caches, and they all load and store data from/to the external (off-chip) main memory. However, the bandwidth to the external memory is much smaller than the SPEs' aggregate bandwidth to the on-chip interconnect bus (EIB) [1]. For applications that frequently access off-chip main memory, such as streaming computations, stream-based sorting, or dataparallel computations on large vectors, the ratio between computational work and memory transfer is low, such that the limited bandwidth to off-chip main memory constitutes the performance bottleneck. This problem will become even more severe with the expected increase in the number of cores in the future. Consequently, the generation of memory-efficient code is an important optimization to consider for such memory-intensive computations.

Scalable parallelization on such architectures should therefore trade increased communication between the SPEs over the high-bandwidth EIB for a reduced volume of communication with external memory, and thereby improve the computation throughput for memory-intensive computations. This results in an *on-chip pipelining* technique: The computations are reorganized such that intermediate results (temporary vectors) are not written back to main memory but instead forwarded immediately to a consuming successor operation. This requires some buffering of intermediate results in on-chip memory, which is necessary anyway in processors like Cell in order to overlap computation with bulk (DMA) communication. It also requires that all tasks (elementary streaming operations) of the algorithm be active simultaneously; tasks assigned to the same SPE will be scheduled round-robin, each with a SPE time share corresponding to its relative computational load. However, as we would like to guarantee fast user-level context switching among the tasks on a SPE, the limited size of Cell's local on-chip memory then puts a limit on the number of tasks that can be mapped to a SPE, or correspondingly a limit on the size of data packets that can be buffered, which also affects performance. Moreover, the total volume of intermediate data forwarded on-chip should be low and, in particular, must not exceed the capacity of the on-chip bus.

We formalize the problem by modeling the application as a weighted acyclic task graph, with node and edge weights denoting computational load and communication rates, respectively, and the Cell processor by its key architectural parameters. We assume that only one application is using

the Cell processor at a time. Task graph topologies occuring in practice are, e.g., complete b-ary trees for b-way merge sort or bitonic sort, butterfly graphs for FFT, and arbitrary tree and DAG structures representing vectorized dataparallel computations. These applications seem to be major application areas for Cell BE besides gaming. On-chip pipelining then becomes a constrained optimization problem to determine a mapping of the task graph nodes to the SPEs that is an optimal or near-optimal trade-off between load balancing, buffer memory consumption, and communication load on the on-chip bus.

To solve this multi-objective optimization problem, we propose an integer linear programming (ILP) formulation that allows to compute Pareto-optimal solutions for the mapping of small to medium-sized task graphs with a state-of-the-art ILP solver. For larger general task graphs we provide a heuristic two-step approach to reduce the problem size. We exemplify our mapping technique with several memory-intensive example problems: with acyclic pipelined task graphs derived from dataparallel code, with complete b-ary merger tree pipelines for parallel mergesort, and with butterfly pipelines for parallel FFT. We validate the mappings with discrete event simulations. Details are given in a forthcoming paper [2].

For special task graph topologies such as merge trees, more problem-tailored solutions can be applied. In previous work [3] on pipelined parallel mergesort, we described a tree-specific divide-and-conquer heuristic and an ILP formulation to compute a good or even optimal placement of the tasks of the resulting tree-shaped pipelined computation. These results can be used to improve Cell-specific merge sort or bitonic sort implementations reported in the literature [4, 5].

In the present paper, we briefly summarize some of our very recent results [2, 3] in this area. In Sect. 2 we present optimal mapping results, and in Sect. 3 we summarize heuristic results for large task graphs. In Sect. 4, we present a new tree-specific approximation algorithm. In Sect. 5 we summarize related work, and Sect. 6 concludes.

2. Optimal Mapping of Task Graphs for On-Chip Pipelining

We start by introducing some basic notation and stating the general optimization problem to be solved. We then give an integer linear programming (ILP) formulation for the problem, which allows to compute optimal solutions for small and middle-sized pipeline task graphs, and report on the experimental results obtained for examples taken from the domain of streaming computations.

Problem definition Given is a set $P = \{P_1, \ldots, P_p\}$ of p processors and a directed acyclic task graph $G = (V, E)$ to be mapped onto the processors. Input is fed at the sources, data flows in direction of the edges, output is produced by the sinks.

Each node (task) v in the graph processes the incoming data streams and combines them into one outgoing data stream. With each edge $e \in E$ we associate the (average) rate $\tau(e)$ of the data stream flowing along e. In all types of streaming computations considered in this work, all input streams of a task have the same rate. However, other scenarios with different τ rates for incoming edges may be possible.

The *computational load* $\rho(v)$ denotes the relative amount of computational work performed by the task v, compared to the overall work $\sum_{v \in V} \rho(V)$ in the task graph. It will be proportional to the processor time that a node v places on a processor it is mapped to. In most scenarios, ρ is proportional to the data rate $\tau(e)$ of its (busiest, if several) output stream e. Reductions are a natural exception here; their processing rate is proportional to the input data rate.

In contrast to the averaged values ρ and τ, the actual computational load (at a given time) is usually depending on the current or recent data rates τ. In cases such as mergesort where the input data rates may show higher variation around the average τ values, also the computational load will be varying when the jitter in the operand supply cannot be compensated for by the limited size buffers.

For presentation purposes, we usually normalize the values of ρ and τ such that the heaviest loaded task r obtains $\rho(r) = 1$ and the heaviest loaded edge e obtains $\tau(e) = 1$. For instance, the root r of a merge tree will have $\rho(r) = 1$ and produce a result stream of rate 1. The computational load and output rate may of course be interpreted as node and edge weights of the task graph, respectively.

The *memory load* $\beta(v)$ that a node v will place on the (SPE) processor it is mapped to (including packet buffers, code, stack space) is usually just a fixed value depending on the computation type of v, because the node needs a fixed amount for its code, for buffering transferred data, and for the internal data structures it uses for processing the data. In homogeneous task graphs such as merge trees or FFT butterflies, all $\beta(v)$ are equal. In this case, we also normalize the memory loads such that each task v gets memory load $\beta(v) = 1$.

We construct a mapping $\mu : V \to P$ of nodes to processors. Under this mapping μ, a processor P_i has *computational load*

$$C_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \rho(v),$$

i.e. the sum of the load of all nodes mapped to it, and it has *memory load*

$$M_\mu(P_i) = \sum_{v \in \mu^{-1}(P_i)} \beta(v)$$

which is $1 \cdot \#\mu^{-1}(P_i)$ for the case of homogeneous task graphs.

The mapping μ that we seek shall have the following properties:

1. The maximum computational load $C^*_\mu = \max_{P_i \in P} C_\mu(P_i)$ among the processors shall be minimized. This requirement is obvious, because the lower the maximum computational load, the more evenly the load is distributed over the processors. With a completely balanced load, C^*_μ will be minimized.

2. The maximum memory load $M^*_\mu = \max_{P_i \in P} M_\mu(P_i)$ among the processors shall be minimized. The maximum memory load is proportional to the number of the buffers. As the memory per processor is fixed, the maximum memory load determines the buffer size. If the buffers are too small, communication performance will suffer.

3. The *communication load* $L_\mu = \sum_{(u,v)\in E, \mu(u)\neq\mu(v)} \tau(u)$, i.e. the sum of the edge weights between processors, should be low.

ILP Formulation We are given a task graph with n nodes (tasks) and m edges, node weights ρ, node buffer requirements β, and edge weights τ. The ILP model for mapping the task graph to a set P of SPEs is summarized in Figure 1; for more details see [2].

For a Cell with p SPEs and a general task graph with n nodes and m edges, our ILP model uses $np + mp = O(np)$ boolean variables, 1 integer variable, 2 linear variables, and $2mp + 2p + 2 = O(np)$ constraints. We implemented the ILP model in CPLEX 10.2 [6], a commercial ILP solver.

By choosing the ratio of ϵ_M to ϵ_C, we can only find two extremal Pareto-optimal solutions, one with least possible *maxMemoryLoad* and one with least possible *commLoad*. In order to enforce finding further Pareto-optimal solutions that may exist in between, one can use any fixed ratio ϵ_M/ϵ_C, e.g. at 1, and instead set a given minimum memory load to spend (which is integer) on optimizing for *commLoad* only:

$$\textit{maxMemoryLoad} \geq \textit{givenMinMemoryLoad}$$

For modeling task graphs of mergesort as introduced above, we generated binary merge trees with k levels (and thus $2^k - 1$ nodes) intended for mapping to $p = k$ processors [3]. Table 1 shows all Pareto-optimal solutions that CPLEX found for $k = p = 5, 6, 7$. While most optimizations for $k = 5, 6, 7$ took just a few seconds, CPLEX hit a timeout after 24 hours for $k = 8$ and only produced approximate solutions with a memory load of at least 37. Figure 2 shows one Pareto-optimal mapping for $k = 5$.

Solution variables:
Binary variables x, z with
$x_{v,q} = 1$ iff node v is mapped on processor q, and
$z_{(u,v),q} = 1$ iff both source u and target v of edge (u, v) are mapped to processor q.
The integer variable *maxMemoryLoad* will hold the maximum memory load assigned to any SPE in P.
The linear variable *maxComputLoad* yields the maximum accumulated load mapped to a SPE.
Constraints:
Each node must be mapped to exactly one processor:

$$\forall v \in V: \quad \sum_{q \in P} x_{v,q} = 1$$

The maximum load mapped to a processor is computed as

$$\forall q \in P: \quad \sum_{v \in V} x_{v,q} \cdot \rho(v) \leq \textit{maxComputLoad}$$

The memory load should be balanced:

$$\forall q \in P: \quad \sum_{v \in V} x_{v,q} \cdot \beta(v) \leq \textit{maxMemoryLoad}$$

Communication cost occurs whenever an edge is not internal, i.e. its endpoints are mapped to different SPEs.

$$\begin{aligned} \forall (u,v) \in E,\ q \in P: \quad & z_{(u,v),q} \leq x_{v,q} \\ & z_{(u,v),q} \leq x_{u,q} \end{aligned}$$

and in order to enforce that a $z_{(u,v),q}$ will be 1 wherever it could be, we have to take up the (weighted) sum over all z in the objective function. This means, of course, that only optimal solutions to the ILP are guaranteed to be correct with respect to minimizing communication cost. We accept this to avoid quadratic optimization, and because we also want to minimize the maximum communication load. The communication load is the communication volume over all edges minus the volume over the internal edges:

$$\textit{commLoad} = \sum_{e \in E} \tau(e) - \sum_{e \in E} \sum_{q \in P} z_{e,q} \cdot \tau(e)$$

Objective function: Minimize

$$\begin{aligned} \Lambda \cdot \textit{maxComputLoad} \ + \ & \epsilon_M \cdot \textit{maxMemoryLoad} \\ & + \epsilon_C \cdot \textit{commLoad} \end{aligned}$$

with Λ chosen large enough to prioritize computational load balancing over all other optimization goals; the positive weights $0 \leq \epsilon_M < 1$ and $0 < \epsilon_C < 1$ are chosen to give preference to *maxMemoryLoad* or *commLoad* as secondary optimization goal.

Figure 1. ILP model for mapping task graphs.

Table 1. The Pareto-optimal solutions for mapping b-ary merge trees, found with ILP, for $b = 2$, $k = p = 5,\ 6,\ 7$.

k	binary variables	constraints	*max. memory load*	*commLoad*
$k = 5$	305	341	8	2.5
			9	2.375
			10	1.75
$k = 6$	750	826	13	2.625
			14	2.4375
			15	1.9375
			20	1.875
$k = 7$	1771	1906	21	2.375
			29	2.3125
			30	2

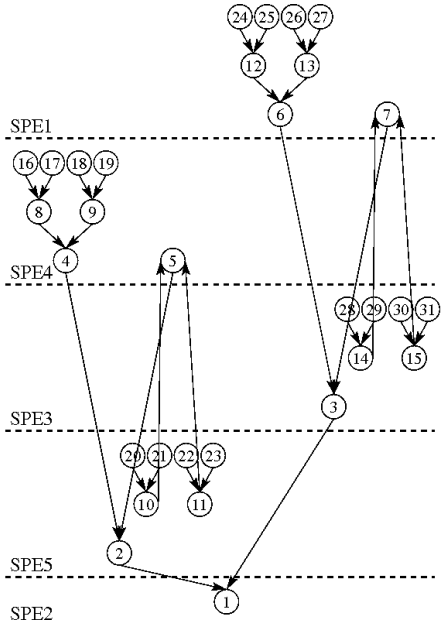


Figure 2. A Pareto-optimal solution for mapping a 5-level merge tree onto 5 processors with maximum memory load 8 (merger tasks on an SPE) and communication load 2.5 (accumulated rate of all inter-SPE edges), computed by the ILP solver.

To test the performance of our merger tree mappings with respect to load balancing, we implemented a discrete event simulation of the pipelined parallel mergesort. The simulation is quite accurate, as the variation in runtime for merger nodes is almost zero, and communication and computation can be overlapped perfectly by mapping several nodes to one SPE. We have investigated several mappings resulting from our mapping algorithm. The 5-level tree of Fig. 2 realizes a 32-to-1 merge. The maximum memory load of 8 merger tasks (needing 5 buffers each) on a SPE still yields a reasonable buffer size of 4 KB, accumulating to a maximum of 160 KB for buffers per SPE. With 32 input blocks of 2^{20} sorted random integers, the pipeline efficiency was 93%. In comparison to the corresponding merge phase in [5], memory bandwidth requirements decreased by a factor of 2.5, but as now 5 instead of 4 SPEs are utilized, this translates to a factor 1.86 in estimated performance gain. For further results for mapped merge trees, see [3].

In order to test the ILP model with dataparallel task graphs, we used several hand-vectorized fragments from the Livermore Loops and synthetic kernels, see Table 2. Such task graphs are usually of very moderate size, and computing an optimal ILP solution for a small number of SPEs takes only a few seconds in most of the cases. For two common Cell configurations ($p = 6$ as in PS3, and $p = 8$), the generated ILP model sizes (after preprocessing) and the times for optimization with memory load preference are given in Table 2. A discrete event simulation of the LL9 mapping on 6 SPEs achieved a pipeline efficiency of close to 100%. Further results and discussion can be found in [2].

3. Heuristic Algorithms for Large Task Graphs

The ILP solver works well for small task graph and machine sizes. However, for future generations of Cell with many more SPEs and for larger task graph sizes, computing optimal mappings with the ILP approach will no longer be computationally feasible. For the case of general task graphs, we developed a divide-and-conquer based heuristic [2] where the divide step uses the ILP model for $p = 2$. An example mapping is given in Fig. 3 for Livermore Loop 9.

4. New Approximation Algorithm for Mapping Merge Trees

We consider the mapping problem for task graphs with the structure of b-ary merger trees, as in b-ary merge sort. In previous work, we presented an approximation algorithm based on divide-and-conquer [3]. Its approximation guarantee for the maximum memory load mainly depends on the tree size k_0 considered as base case in the recursive solution (which is, e.g. solved optimally by the ILP method); the worst-case maximum memory load is by a factor k/k_0 larger than a straightforward lower bound (see Lemma 1), but the quality is much better in practice [3]. As an example, for $k = p = 5$ and $b = 2$ (i.e., a 32-to-1 binary merger tree), the resulting mapping μ_1 is different from the Pareto-optimal one shown in Figure 2 but has the same quality (optimal maximum memory load 8 and communication

Table 2. ILP models for dataparallel task graphs extracted from the Livermore Loops (LL) and from some synthetic kernels.

Kernel	Description	#Nodes	#Edges	ILP model for $p = 6$			ILP model for $p = 8$		
		n	m	var's	constr.	time	var's	constr.	time
LL9	Integrate predictors	28	27	333	371	2:07s	443	485	—
LL10	Difference predictors	29	28	345	384	0:06s	459	502	1:26:39s
LL14	1D particle in cell, 2nd loop	19	21	243	290	0:03s	323	380	1:05s
LL22	Planckian distribution	10	8	111	125	<0:01s	147	163	<0:01s
FIR8	8-tap FIR filter	16	22	231	299	45:04s	307	393	0:04s
T-8	Binary tree, 16 leaves	31	30	369	410	5:36s	491	536	0:11s
C-6	Cook pyramid, 6 leaves	21	30	309	400	27:56s	411	526	3:22s

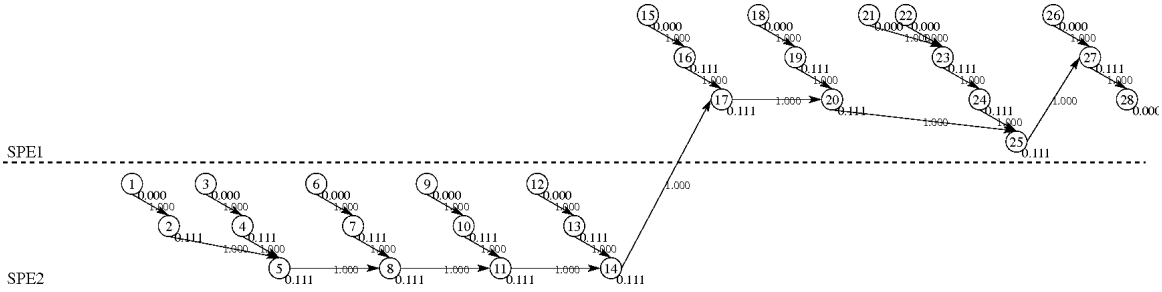


Figure 3. ILP solution for partitioning the task graph of Livermore Loop 9 into two (thus $p = 2$) subgraphs, each to be mapped separately on a Cell SPE subset of size 4.

load 2.5). The discrete event simulation reported also for this mapping a pipeline efficiency of 93%.

In the following, we give an alternative approximation algorithm where the maximum memory load is by a factor at most b larger than the lower bound, independent of the size of the tree, i.e. the number of levels.

We will use the notations from Sect. 2. Our task graph T is a complete and balanced b-ary k-level merge tree. As each task has exactly one outgoing edge, we identify the rate τ of edges with the computational load of their origin node. Thus $\tau(v, w) = \rho(v)$. As task v merges the b incoming data streams into one outgoing data stream with rate $\tau(v)$, the incoming data streams on average will have rate $\tau(v)/b$, if we assume only finite buffering within nodes. With normalization, the tree root r will have $\rho(r) = 1$, and thus each node v on level i of the tree, where $0 \leq i \leq k - 1$, has $\rho(v) = b^{-i}$ on average.

We extend ρ and τ to subgraphs of the merge tree. A subgraph's computational load is the sum its node loads, and its outgoing data rate is the aggregate rate of all edges leaving the subgraph. For example, a subtree of l levels rooted at v has computational load $l \cdot \rho(v)$ and data rate $\tau(v)$.

The mapping μ for T that we seek shall have the properties 1–3 already listed for the general case in Section 2, but in addition, it shall also fulfill:

4. As often as possible, sibling nodes (nodes u and v with a common successor w, i.e. where $(u, w) \in E$ and $(v, w) \in E$) should be mapped to the same processor.

   Note that a merger should deliver merged data buffers at an actual output rate that does not significantly fall short of the average output rate, because otherwise the preceding and subsequent mergers may be delayed, too, due to limited buffer capacity. A drop in the output rate may be caused by phases of unequal distribution of data in the input sequences, such that a merger processes, in such a phase, mainly input data coming from one subtree only, which effectively stalls the other subtree(s). Short phases can be caught by buffering (if buffers are sufficiently large) and have thus no effect, while long phases may lead to idle times on some processors. If sibling merger nodes are mapped to the same processor, such a stall of a sibling node allows to temporarily give an accordingly larger processor time share to the busier sibling(s), maintaining a more balanced overall output rate of the siblings towards the common parent node.

Lemma 1 (Lower bounds) *In any mapping μ the maximum computational load is at least k/p, and the maximum memory load is at least $(b^k - 1)/((b - 1)p)$.*

Proof: As there are b^i nodes in level i, each with com-

putational load b^{-i}, the computational load in each level equals 1, i.e. the load of the k-level tree equals k. As this load is spread over p processors, there will be at least one processor with computational load at least k/p.

As there are $(b^k-1)/(b-1)$ nodes in a k-level balanced b-ary tree, each with memory load c, the memory load of the tree equals $c \cdot (b^k-1)/(b-1)$. As this load is spread over p processors, there will be at least one processor with memory load at least $(b^k-1)/((b-1)p)$. ∎

Construction Consider as a first try the case $p = k$ and the mapping μ_0 that maps all nodes of level i onto processor P_i. Obviously, this mapping fulfils properties 1 and 3, as the computational load of each level equals 1 (see Lemma 1), and as siblings in the tree are always on the same level and hence mapped to the same processor. However, b^{k-1} nodes of level $k-1$ are mapped to processor P_{k-1}, and hence $M^*_{\mu_0} = c \cdot b^{k-1}$ and thus a factor of about $k/2 \leq k(b-1)/b \leq k$ away from the lower bound of Lemma 1. This restriction is serious, as each processor only contains a fixed amount of local memory, so that either, when we consider the memory load of each task to be fixed, the maximum number k to which this mapping scales is severely limited. If we do not fix the memory load of the task, the memory available for each node is—at least on level $k-1$, i.e. for at least half of all nodes because of $b \geq 2$—shrinking by a factor of k faster with growing k than necessary, i.e. buffer size will soon become very small, which also affects performance of data transfer.

We therefore devise a mapping μ_1 that is constructed in several steps, and in each step i maps l_i levels of the tree onto l_i processors. Let $k_0 = k$ be the number of levels and processors in step 0. In step i, if $k_i \geq 2$, we map $l_i \leq k_i - 1$ of the k_i levels, starting from the leaves, onto a respective number of processors, so that $k_{i+1} = k_i - l_i$ levels and processors remain. If $k_i = 1$, we map the tree root onto the last processor, and the mapping is complete. As each level of the tree has a computational load of 1, the mapping must be such that each processor receives a load of 1 to minimize $C^*_{\mu_1}$.

We choose l_i to be the largest power of b less than or equal to $k_i - 1$. The l_i levels then consist of $b^{k_{i+1}}$ balanced b-ary trees of l_i levels each. If $l_i \leq b^{k_{i+1}}$, then l_i divides $b^{k_{i+1}}$ because it is also a power of b, and we map $b^{k_{i+1}}/l_i$ trees on each of the processors. This balances both maximum computational and maximum memory load.

The case $l_i > b^{k_{i+1}}$ is illustrated in Fig. 4. In this case, we can write $l_i = b^x \cdot b^{k_{i+1}}$, where $x \geq 1$ is an integral number. In this case, we define $l'_i = l_i - b^x$ and first map the l'_i levels starting from the leaves. Those levels consist of $b^{k_{i+1}+b^x}$ balanced b-ary trees of l'_i levels each. As $b^x \geq x$ because of $b \geq 2$ and $x \geq 1$, it follows that $b^{k_{i+1}+b^x} \geq b^{k_{i+1}+x} = l_i$ and that this number is even an

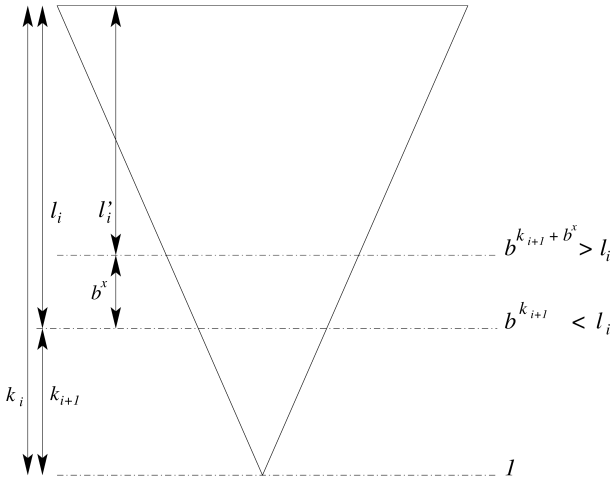


Figure 4. Step i of the construction of mapping μ_1.

integral multiple of l_i because l_i is also a power of b. Thus, we can map the trees of the last l'_i levels evenly onto the l_i processors. For the remaining b^x levels to be mapped in this step, we map those levels starting with the level closest to the root having $b^{k_{i+1}}$ nodes: we map each node onto one processor, using $b^{k_{i+1}}$ processors. For the next level having $b \cdot b^{k_{i+1}}$ nodes, we map b nodes on each processor, using another $b^{k_{i+1}}$ processors. When we have finished with those b^x levels, we have used $b^x \cdot b^{k_{i+1}} = l_i$ processors. Note that this straightforward placement corresponds to applying mapping μ_0 for the $b^{k_{i+1}}$ trees of $k = b^x$ levels each, with the processor capacity scaled down to $b^{-k_{i+1}}$. We might also apply mapping μ_1 recursively to further balance the load.

On each processor, we have placed a load of $l'_i/l_i = 1 - b^{-k_{i+1}}$ by mapping l'_i levels, and $b^{-k_{i+1}}$ by mapping the first b^x levels. It follows that the computational load on each processor is 1. The maximum memory load is determined in step $i = 0$, because the majority of the nodes is mapped there. In this step $(b^k - b^{k-l_0})/(b-1)$ nodes are mapped onto l_0 processors, so that each processor receives a memory load of

$$\frac{b^k - b^{k-l_0}}{(b-1)l_0} < b \cdot \frac{b^k - 1}{(b-1)k}$$

because $l_0 \geq k/b$. Thus, the memory load is larger than the lower bound by a factor less than b. Note that this is not completely exact because the b^x levels — if they are used in the first step — are not mapped with a completely even memory load. However, the imbalance is only very slight, as our simulations will show.

In each step, there are at most two levels (the first one

of the b^x and the first one of the l'_i) where siblings are not placed on the same processor.

As in each step i the largest power of b less than k_i is chosen as the number l_i of levels mapped, the number r of steps made by the mapping algorithm is one plus the cross sum of $k-1$ in b-ary representation, and thus $r \leq 1 + (b-1) \cdot \log_b(k-1)$.

We summarize the properties of mapping μ_1:

Lemma 2 *The maximum computational load of mapping μ_1 is $C^*_{\mu_1} = 1$, which is optimal.*

The maximum memory load of mapping μ_1 is about $M^*_{\mu_1} = \frac{b^k - b^{k-l_0}}{(b-1)l_0}$, which is larger than the lower bound by a factor of less than b.

In at least $k - 2r$ levels, siblings are mapped to the same processor, where $r \leq 1 + (b-1)\log_b(k-1)$ is the number of the steps in the construction of the mapping.

We illustrate the mapping algorithm for the case $b = 2$ and $k = 5$. The resulting mapping is identical to the one in Fig. 2, i.e. the approximation algorithm produces an optimal result. In step $i = 0$, we have $l_0 = 4$ as this is the largest power of 2 less than $k_0 = k = 5$. Hence, $k_1 = 1$. The levels to be mapped consist of $2^{k_1} = 2$ trees of 4 levels, and thus cannot be mapped directly. It follows that $x = 1$ as $l_0 = 4 = 2^1 \cdot 2^1 = 2^{k_1} \cdot 2^x$, and thus $l'_0 = l_0 - 2^x = 2$. The last two levels of the 5-level tree consist of 8 trees, so that two of them are mapped onto each processor. Then we place the remaining $2^x = 2$ levels, of which the first consists of two nodes, that are mapped onto two processors, one node on each processor. The last level consists of 4 nodes, of which 2 are mapped on each of two processors. Finally, in step $i = 1$, we have $k_1 = 1$ and map the root onto the last processor. The maximum memory load of the mapping is 8 which is optimal (see previous section) although it is a factor of 1.29 away from the lower bound.

As a second example we consider $k = 8$ and $b = 2$. In this example $b^{k_{i+1}} \geq l_i$ for all steps i. In step 0, we map $l_0 = 4$ levels of the tree onto 4 processors, in step 1 we map $l_1 = 2$ levels, and in steps 2 and 3 we map 1 level, respectively. The resulting mapping is depicted in Fig. 5. The maximum computational load on each processor is 1, which is optimal, and the maximum memory load is 60, on processors 0 to 3, which is a factor of 1.9 away from the lower bound.

Both examples were chosen in part because they represent two extremes: $k = 5 = 2^2 + 1$ is a power of two plus one, and thus l_0 can be chosen the maximum value so that $k_1 = 1$, and the mapping can be constructed in two steps. The closer l_0 is to k, also the closer the maximum memory load is to the lower bound. In contrast, for $k = 8$, we must choose $l_0 = 4$ which is only half of k, and thus the worst value possible. As a consequence the maximum memory load is by a factor of 1.88 larger than the lower bound.

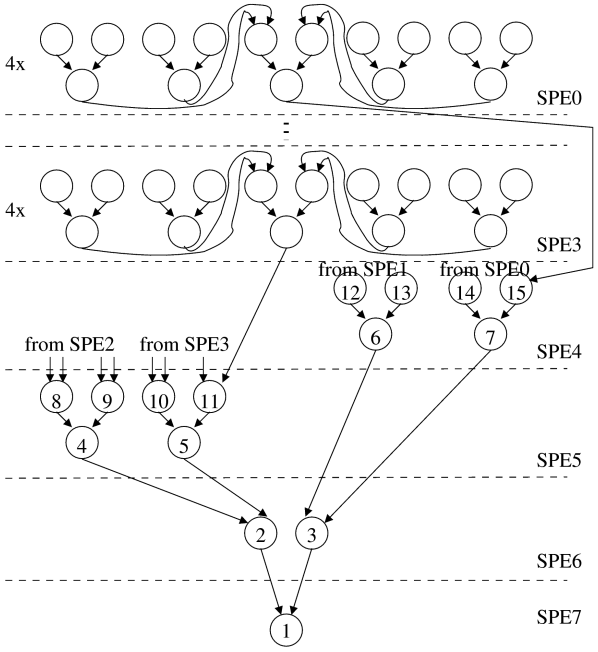


Figure 5. Mapping a 8-level binary tree onto 8 processors.

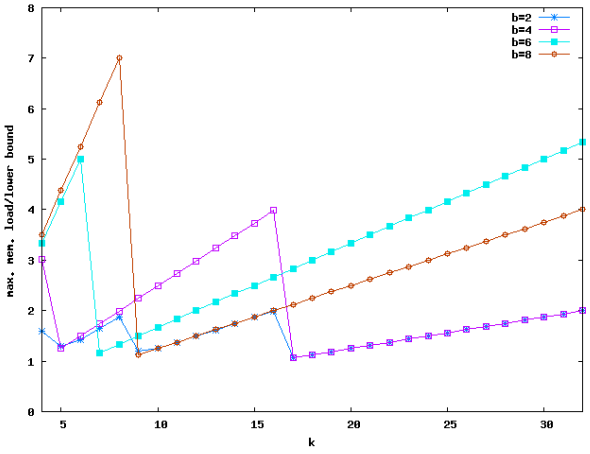


Figure 6. Ratio between max. memory load and lower bound depending on k and b.

In general, the ratio between maximum memory load and lower bound increases with increasing k in intervals $[b^j + 1, \ldots, b^{j+1}]$ from 1 to b. We have illustrated this for $k = 4, \ldots, 32$ and $b = 2, 4, 6, 8$ in Fig. 6.

Several cases remain to be considered. In the case that $p < k$, there are several possibilities. If k is a multiple of p, then we could first construct a mapping onto k pseudo-

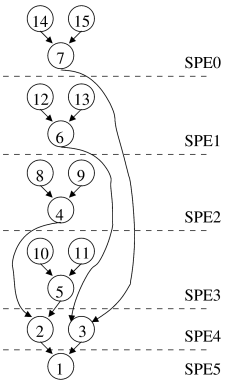


Figure 7. Mapping a 4-level tree onto 6 SPEs.

processors, and then map k/p of those pseudo-processors onto one processor. Alternatively, one could split the k-level tree into p-level sub-trees that can be mapped as before, and have the system work in a scheduled way, e.g. round-robin, on the sub-trees. If k is not a multiple of p, then we again split the tree into p-level sub-trees starting from the leaves. The sub-tree containing the root will contain only $k \bmod p < p$ levels, a case which is treated below.

Note that k is an input parameter of our problem as the sorting algorithm in principle is free to choose k as seems suitable. Therefore k should be chosen such it fits well with the available number of processors, i.e. in most cases one will try to choose $k = p$.

In the case that $p > k$, there are again several possibilities. If p is a multiple of k, then one could first construct a mapping onto k pseudo-processors, and then distribute the work of each pseudo-processor evenly onto p/k processors. Note that distributing the work of the pseudo-processor to which the root node is mapped consists of parallelizing a single merge node. However, there are algorithms known in the literature for that problem, see e.g. Chapter 4.2 of [7] that presents a parallel merge on n processors with $O(n \log n)$ work.

In case that p is only slightly larger than k, one may also think of distributing the work of processors to which the leaves are mapped, onto several processors. The reduction of the computational load for these processors takes into account that the computational load is an *average* over the time, and can compensate variations too large to be taken care of by buffering. An example is depicted in Fig. 7 for $k = 4$ and $p = 6$. The mapping for $k = p = 4$ uses $l_0 = 2$ in the first step, but distributes the load for two processors now onto four processors.

So far, we have only considered the memory load, and not the load on the ring network. The processor P_{p-1} holding solely the root node will have an output rate of 1, which is transported over the ring to the external memory. Each set of processors maps a number of levels. The rate of the communication leaving those processors sums up to 1 as well, as this is the load on each level. Hence, no part of the ring will have a higher network load, so that also the network load scales well with the algorithm.

5. Related Work

Partitioning and mapping of task graphs is, in general, a NP-complete problem and has been discussed a lot in the literature.

One application area is, as in our case, the parallelization of programs with given dependence graph for execution on a (mostly, shared memory) parallel computer, with the objective to balance the work load of the partitions, minimize the number of partitions (aka. *processor minimization*), and/or minimize the overall weight of all edges cut by the partitioning, as all these are supposed to correspond to expensive shared memory accesses (aka. *bandwidth minimization*).

Another related area is the (spatial) clustering of logic circuits into partitions each matching a maximum chip size constraint, while the communication between partitions must fit an upper limit on the number of pins per chip. Here, one is (as in our case) mainly interested in reducing the accumulated weight of all edges cut between any two adjacent partitions (aka. *bottleneck minimization*).

There is a wealth of literature on mapping and scheduling acyclic task graphs of streaming computations to multiprocessors. Some methods are designed for special topologies, such as linear chains and trees, while others address general task graphs.

Mapping of special topologies For *tree-shaped task graphs*, various partitioning algorithms have been proposed.

Bokhari [8] considers partitioning of trees for master-slave (there called host-satellite) systems where the partition containing the root is mapped to the master (host) processor while the slaves (satellites) are each assigned exactly one complete subtree that is connected directly to the master partition.

Ray and Jiang [9] show that the bandwidth minimization problem is NP-complete even for trees, and give a fast heuristic algorithm for it. In a follow-up paper [10], the same authors give polynomial-time greedy algorithms for bottleneck minimization and processor minimization of tree task graphs.

Most approaches for tree partitioning are for non-pipelined trees and therefore assume that the tree partitions should be connected components (i.e., contiguous subtrees) and exactly one partition be mapped to one processor. This does not apply in our case, where partitions can consist of multiple disconnected subtrees, so that processors could be better "filled up" to their computational capacity with residual tree fragments if this improves system throughput. Also,

in our scenario the b-ary tree is always complete, thus we can exploit symmetry properties that are not given in the more general case.

Lüling et al. [11] consider the problem of mapping a tree that evolves in a search problem onto a distributed memory parallel computer in such a way that computation and communication times both are minimized. They focus on trees that evolve dynamically, i.e. are not known beforehand as in our case. The work associated with each tree node seems to be constant while the computational load in our case depends on the tree level of the node. As the tree is not kept completely, memory load plays a minor role. In contrast, we map a tree to be kept completely in memory. Finally, the trees considered in search problems typically are far from balanced and their degree is irregular, while we consider balanced b-ary trees.

Middendorf et al. [12] consider non-pipelined, tree-like task graph structures such as reduction trees, task graphs for parallel prefix computations and *Butterfly graphs*, under the LogP cost model that accounts for transfer latency and limited communication bandwidth in message passing systems. They give polynomial-time algorithms for computing optimal schedules for special cases. However, memory constraints or pipelined versions of these task graphs are not considered.

Mapping of general task graphs The approaches for mapping general task graphs can be roughly divided into two classes: Non-overlapping scheduling and overlapping scheduling.

Non-overlapping scheduling schedules a single execution of the program (and repeats this for further input sets if necessary); it aims at minimizing the makespan (execution time for one input set) of the schedule, which depends strongly on task and communication latencies, while memory constraints are usually a non-issue here. A typical result is that all tasks on a critical path are mapped to the same processing unit. The mapping and scheduling can thus be done by classical list-scheduling based approaches for task graph clustering that attempt to minimize the critical path length for a given number of processors. Usually, partitions are contiguous subgraphs. The problem complexity can be reduced heuristically by a task merging pre-pass that coarsens the task granularity. Optimization methods applied include e.g. gradient search as in Sarkar and Hennessy [13]. See [14] for a recent survey and comparison.

Szymanek and Kuchcinski [15] propose a heuristic method for memory-aware assignment and scheduling of a task graph to a bus- or link-connected set of processing units. Tasks are parametrized in their code and data memory needs, and edges between tasks by the buffer space requirements on sender and receiver side during the whole communication period that results if an edge is selected as communication edge between partitions. Based on initial estimations for maximum data memory use, this iterative optimization method toggles between two strategies for assignment and scheduling, namely critical path scheduling (which optimizes for the makespan) and scheduling for minimization of memory usage, trying to balance execution time and memory utilization of the resulting solution.

Overlapping scheduling, which is closely related to *software pipelining* [16, 17] and *systolic parallel algorithms* [18], instead overlaps executions for different input sets in time and attempts to maximize the throughput in the steady state, even if the makespan for a single input set may be long. Mapping methods for such pipelined task graphs, especially for signal processing applications in the embedded systems domain, have been described e.g. by Hoang and Rabaey [19] and Ruggiero *et al.* [20]. Our method also belongs to this second category.

Hoang and Rabaey [19] work on a hierarchical task graph such that task granularity can be refined by expanding function calls or loops into subtasks as appropriate. They provide a heuristic algorithm based on greedy list scheduling for simultaneous pipelining, parallel execution and retiming to maximize throughput. The resulting mapped pipeline is a linear graph where each pipeline stage is assigned one or several processors. Buffer memory requirements are considered only when checking feasibility of a solution, but are not really minimized for. The method only allows contiguous subDAGs to be mapped to a processor.

Ruggiero *et al.* [20] decompose the problem into mapping (resource allocation) and scheduling. The mapping problem, which is close to ours, is solved by an integer linear programming formulation, too, and is thus, in general, not constrained to partitions consisting of contiguous subDAGs as in most other methods. Their framework targets MPSoC platforms where the mapped partitions form linear pipelines. Their objective function for mapping optimization is minimizing the communication cost for forwarding intermediate results on the internal bus. Buffer memory requirements are not considered.

6. Conclusion

We have shown how to lower memory bandwidth requirements in code for the Cell BE by on-chip pipelining of memory-intensive computations. To realize pipelining with maximum throughput while reducing on-chip memory load and interprocessor communication, we formulated a general optimization problem for mapping task graphs. We have demonstrated our model with case studies from data-parallel code generation and merge trees in sorting. Small to medium sized problem instances can be solved optimally by ILP, larger ones by heuristics and approximation algorithms. We have also presented a new tree-specific approx-

imation algorithm for the mapping problem.

Implementing and evaluating the resulting code on Cell is an issue of current and future work. The method could be used e.g. as an optimization in code generation for data-parallel code in an optimizing compiler for Cell, such as [21].

Acknowledgements C. Kessler acknowledges partial funding by Vetenskapsrådet, SSF, Vinnova, and CUGS.

References

[1] Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation—a performance view. IBM J. Res. Devel. **51**(5) (Sept. 2007) 559–572

[2] Kessler, C.W., Keller, J.: Optimized mapping of pipelined task graphs on the Cell BE. In: Proc. 14th Int. Workshop on Compilers for Parallel Computing (CPC-2009), Zürich, Switzerland. (January 2009)

[3] Keller, J., Kessler, C.W.: Optimized pipelined parallel merge sort on the Cell BE. In: Proc. 2nd Workshop on Highly Parallel Processing on a Chip (HPPC-2008) at Euro-Par 2008, Gran Canaria, Spain. (2008)

[4] Gedik, B., Bordawekar, R., Yu, P.S.: Cellsort: High performance sorting on the Cell processor. In: Proc. 33rd Int.l Conf. on Very Large Data Bases. (2007) 1286–1207

[5] Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In: Proc. 16th Int.l Conf. on Parallel Architecture and Compilation Techniques (PACT), IEEE Computer Society (2007) 189–198

[6] ILOG Inc.: CPLEX v. 10.2. www.ilog.com (2007)

[7] JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley (1992)

[8] Bokhari, S.H.: Partitioning problems in parallel, pipelined and distributed computing. IEEE Transactions on Computers **37**(1) (January 1988)

[9] Ray, S., Jiang, I.: Improved algorithms for partitioning tree and linear task graphs on shared memory architecture. In: Proceedings of the 14th International Conference on Distributed Computing Systems. (June 1994) 363–370

[10] Ray, S., Jiang, I.: Sequential and parallel algorithms for partitioning tree task graphs on shared memory architecture. In: Proc. International Conference on Parallel Processing, Volume 3. (August 1994) 266–269

[11] Lüling, R., Monien, B., Reinefeld, A., Tschöke, S.: Mapping tree-structured combinatorial optimization problems onto parallel computers. In: Solving Combinatorial Optimization Problems in Parallel - Methods and Techniques, London, UK, Springer-Verlag (1996) 115–144

[12] Middendorf, M., Löwe, W., Zimmermann, W.: Scheduling inverse trees under the communication model of the LogP-machine. Theoretical Computer Science **215** (1999) 137–168

[13] Sarkar, V., Hennessy, J.: Compile-time Partitioning and Scheduling of Parallel Programs. In: Proc. ACM SIGPLAN Symp. on Compiler Construction. (1986) 17–26

[14] Kianzad, V., Bhattacharyya, S.S.: Efficient techniques for clustering and scheduling onto embedded multiprocessors. IEEE Trans. on Par. and Distr. Syst. **17**(7) (July 2006) 667–680

[15] Szymanek, R., Kuchcinski, K.: A constructive algorithm for memory-aware task assignment and scheduling. In: CODES '01: Proc. 9th int. symposium on Hardware/software codesign, New York, NY, USA, ACM (2001) 147–152

[16] Rau, B., Glaeser, C.: Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: Proc. 14th Annual Workshop on Microprogramming. (1981) 183–198

[17] Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In: Proc. ACM SIGPLAN Symp. on Compiler Construction. (July 1988) 318–328

[18] Kung, H.T.: Why systolic architectures? IEEE Computer **15** (January 1982) 37–46

[19] Hoang, P.D., Rabaey, J.M.: Scheduling of DSP programs onto multiprocessors for maximum throughput. IEEE Trans. on Signal Processing **41**(6) (June 1993) 2225–2235

[20] Ruggiero, M., Guerri, A., Bertozzi, D., Milano, M., Benini, L.: A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness. Int. J. of Parallel Programming **36**(1) (February 2008)

[21] Eichenberger et al., A.E.: Using advanced compiler technology to exploit the performance of the Cell Broadband Engine (TM) architecture. IBM Systems Journal **45**(1) (2006)

Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms

Håkan Lundvall, Kristian Stavåker, Peter Fritzson, Christoph Kessler
PELAB – Programming Environments Laboratory
Dept. of Computer and Information Science
Linköping University, SE-581 83 Linköping, Sweden
{haklu, krsta, petfr, chrke}@ida.liu.se

Abstract

In this work we report results from a new integrated method of automatically generating parallel code from Modelica models by combining parallelization at two levels of abstraction. Performing inline expansion of a Runge-Kutta solver combined with fine-grained automatic parallelization of the right-hand side of the resulting equation system opens up new possibilities for generating high performance code, which is becoming increasingly relevant when multi-core computers are becoming commonplace. An implementation, in the form of a backend module for the OpenModelica compiler, has been developed and used for measurements on two architectures: Intel Xeon and SGI Altix 3700 Bx2. This paper also contains some very recent results of a prototype implementation of this parallelization approach on the Cell BE processor architecture.

Keywords: Modelica, automatic parallelization, equation-based modeling.

1. Introduction

Equation-based Object-Oriented (EOO) modeling languages (see Section 2), such as Modelica [12][13] and VHDL-AMS [5], are used for modeling and simulation of increasingly complex industrial applications, requiring higher performance of hardware architectures used. To reach acceptable levels of performance when simulating these models, increased usage of parallel architectures, especially multi-core ones, will be necessary. In this context, there is a great motivation for exploring automatic and partly manual methods to extract parallelism from mathematical models. Several approaches are briefly described in Section 3.

In this work we extend previous approaches. We present a method [1] of automatically generating parallel code from Modelica models by combining parallelization at two levels of abstraction. Our work represents a new way of automatically detecting pipelining possibilities in the total task graph (generated from the simulation problem) containing both the solver stages (an inline expansion of a Runge-Kutta solver) and the right hand side of the system, and automatically generating parallelized code optimized for the specific parameters of the target machine. This is a continuation of the work in [6].

We have introduced a new way of scheduling the task graph generated from the simulation problem which utilizes knowledge about locality of the simulation problem and generates a computation pipeline such that processors early in the pipeline can carry on with subsequent time steps while the end of the pipeline still computes the current step

If the model in question contains algebraic loops the evaluation of the right hand side will involve solving a system of simultaneous equations. If this system involves non-linear equations, it is solved using an iterative solver. From the scheduler's point of view such a set of equations is treated as one, very expensive, task, so that the generated task graph is always an acyclic graph. This means that models containing large algebraic loops are not suitable for this pipelining approach. Our parallelization approach has not yet been adapted to hybrid simulation problems.

We report implementation details and measurements for three different hardware configurations: Intel Xeon, SGI Altix 3700 Bx2 and Cell BE. Most of this paper, except the very recent results on the Cell BE processor, has recently been accepted for publication [2].

The paper has the following structure. In Section 2 we present some background information on mathematical modeling languages, especially Modelica and its open-source implementation OpenModelica. Section 3 contains some background information on methods for exploiting parallelism in mathematical models. Section 4 introduces methods for combining parallelism extracted from several levels of abstraction, leading over to the new work presented at the end of that section, and in Sections 5 to 8. Finally, Sections 9 and 10 present our conclusions and future directions of our work.

2. Background on Mathematical Modeling and Modelica

Modelica is a rather new language for equation-based object-oriented mathematical modeling and is being developed through an international effort [12][13]. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a de facto standard. It allows defining simulation models in a declarative

manner, modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism. The multi-domain capability of Modelica gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model.

In the context of Modelica class libraries software components are Modelica classes. However, when building particular models, components are instances of those Modelica classes. Classes should have well-defined communication interfaces, sometimes called ports, in Modelica called connectors, for communication between a component and the outside world. A component class should be defined independently of the environment where it is used, which is essential for its reusability. This means that in the definition of the component including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, is allowed. A component may internally consist of other connected components, i.e., hierarchical modeling.

Compiling a Modelica model involves going through several stages as shown in Figure 1.

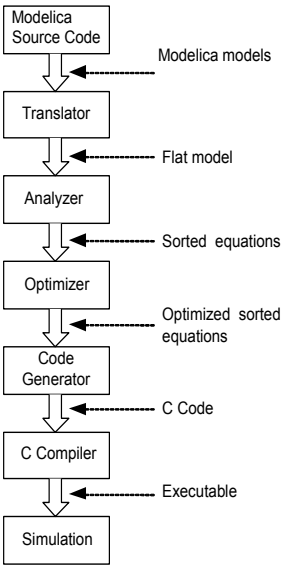


Figure 1. OpenModelica compiler translation stages.

First the hierarchy imposed by the object oriented and component based modeling style is flattened out to a set of equations and variables. The result of the first stage is what we call a flat model which is then analyzed in order to sort the equations topologically according to data dependencies between the equations. In the Optimizer stage some symbolic manipulations are performed while removing trivial equations and performing symbolic index reduction. Finally, the code generator produces C/C++ code which can be compiled and linked together with a numeric solver and executed.

To summarize, the key characteristics of Modelica are:

- *Object-oriented mathematical modeling*. This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.
- *Acausal modeling*. Modeling is based on equations instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases reusability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation. However, for interfacing with traditional software, algorithm sections with assignments as well as external functions/procedures are also available in Modelica.
- *Physical modeling of multiple application domain*s. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. In Modelica the structure of the model becomes more natural in contrast to block-oriented modeling tools. For application engineers, such "physical" components are particularly easy to combine into simulation models using a graphical editor.

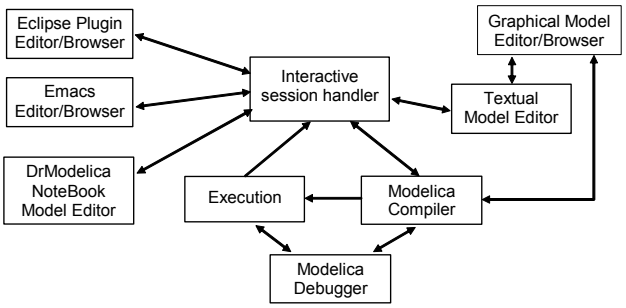


Figure 2. The architecture of the OpenModelica environment.

2.1. The OpenModelica Open Source Environment

The OpenModelica environment is the currently major Modelica open-source tool effort [11] consisting of several interconnected subsystems, as depicted in Figure 2. Arrows

denote data and control flow. Several subsystems provide different forms of browsing and textual editing of Modelica code.

In this research project two parts of the OpenModelica environment are used.

- A Modelica compiler subsystem (called OMC – for Open-Modelica Compiler) translating Modelica to C/C++ code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries.
- An execution and run-time module. This module currently executes compiled binary code from translated expressions and functions, as well as simulation code from equation based models, linked with numerical solvers.

2.2. General Mathematical Description of Simulation Problem

As mentioned in the section 1, in our work we have not yet approached hybrid simulation problems. The compilation and transformation of a plain continuous Modelica model generally results in a Differential Algebraic Equation system (DAE). A DAE system in implicit form can be expressed as follows:

$$0 = g(t, X'(t), X(t), Y(t), P), \quad X(t_0) = X_0$$

Here X is a vector of state variables, X' is a vector of derivatives of the state variables, Y is a vector of algebraic variables, P is a vector of parameters and/or constant, X_0 is a vector of start values and t is the time variable.

Usually such a system is transformed to an Ordinary Differential Equation system (ODE). Such an ODE system can be described as:

$$X'(t) = f(t, X(t), Y(t), P), \quad X(t_0) = X_0$$

An iteration scheme, such as Runge-Kutta, is then applied to this system. When we from now on talk about the "right-hand side" it is f in the above equation system that we refer to. See, for example, [15] for more details on continuous system simulation.

3. Approaches to Exploiting Parallelism in Mathematical Models

There are several approaches to exploit parallelism in mathematical models. In this section we briefly review three main approaches that we are investigating in the context of parallel simulation of equation-based models.

3.1. Automatic Fine-Grained Parallelization of Mathematical Models

One obstacle to parallelization of traditional numerical simulation codes is the prevalence of low-level implementation details in such codes, which also makes automatic parallelization hard.

Instead, it would be attractive to directly extract parallelism from the high-level mathematical model, or from the numerical method(s) used for solving the problem. Such parallelism from mathematical models can be categorized into three groups:

- *Parallelism over the method*. One approach is to adapt the numerical solver for parallel computation, i.e., to exploit parallelism over the method. For example, by using a parallel ordinary differential equation (ODE) solver that allows computation of several time steps simultaneously. However, at least for ODE solvers, only limited parallelism is available. An adoption of alternative solver algorithms with better parallelizability may, in turn, decrease the numerical stability, which could lead to increased simulation time again.
- *Parallelism over time*. A second alternative is to parallelize the simulation over the simulated time. This is however best suited for discrete event simulations, since solutions to continuous time dependent equation systems develop sequentially over time, where each new solution step depends on the immediately preceding steps.
- *Parallelism of the system*. This means that the modeled system (the model equations) is parallelized. For an ODE or DAE equation system, this means parallelization of the right-hand sides of such equation systems which are available in explicit form; moreover, in many cases implicit equations can automatically be symbolically transformed into explicit form.

A thorough investigation of the third approach, automatic parallelization over the system, has been done in our previous work on automatic parallelization (fine-grained task-scheduling) of Modelica-generated simulation code [9][10], see Figure 5.

3.2. Coarse-Grained Explicit Parallelization Using Computational Components

Automatic parallelization methods have their limits. A natural idea for improved performance is to structure the application into computational components using strongly-typed communication interfaces.

This involves generalization of the architectural language properties of Modelica, currently supporting compo-

nents and strongly typed connectors, to distributed components and connectors [17][18]. This will enable flexible configuration and connection of software components on multiprocessors, clusters, or on computational grids, and involves a structured system of distributed solvers/ or solver components.

3.3. Explicit Parallel Programming

The third approach is providing general easy-to-use explicit parallel programming constructs within the algorithmic part of the modeling language. We have previously explored this approach with the NestStepModelica language [3][7]. NestStep is a parallel programming language based on the BSP (Bulk-Synchronous Parallel) model, which is an abstraction of a restricted message passing architecture and charges cost for communication. It is defined as a set of language extensions which, in the case of NestStepModelica, is added to the algorithmic part of Modelica. The added constructs provide shared variables and process coordination. NestStepModelica processes run, in general, on different machines that are coupled by the NestStepModelica language extensions and runtime system to a virtual parallel computer.

4. Combining Parallelism from Several Levels

Models described in equation-based object-oriented modeling languages like Modelica give rise to large differential algebraic equation systems that can be solved using numerical DAE or ODE-solvers. Many scientific and engineering problems require a lot of computational resources, particularly if the system is large or if the right hand side is complicated and expensive to evaluate. Obviously, the ability to parallelize such models is important, if such problems are to be solved in a reasonable amount of time.

As mentioned in the introduction, parallelization of object oriented equation-based simulation code can be done at several different levels. In recent work [6] we started to explore the combination of the following two parallelization approaches:

- *Parallelization across the method*, e.g., where the stage vectors of a Runge-Kutta solver can be evaluated in parallel within a single time step
- *Fine grained parallelization across the system* where the evaluation of the right hand side of the system equations is parallelized.

In this work, we develop an integrated automatic two-level parallelization approach, also including software pipelining. This is a further development of the basic parallel pipelining technique for Runge-Kutta solvers described by Korch and Rauber [8].

We automatically detect pipelining possibilities in the total task graph, which is obtained by inlining the solver stages in the code evaluating the right hand side of the system, and automatically generate parallelized code optimized for the specific latency and bandwidth parameters of the target machine.

5. Pipelining the Task Graph

Since communication between processors is more frequent using this approach we want to make sure that the communication interferes as little as possible with the computation. Therefore, we schedule the tasks in such a way that communication taking place within the same simulation step is always directed from a processor with lower rank to a higher ranked processor. In this way the lower ranked processor is always able to carry on with calculations even if the receiving processor temporarily falls behind. Results needed in the next simulation step by other processors are calculated and sent out first so that a lower ranked processor can carry on with the next time step as early as possible. This scheduling scheme is depicted in Figure 4 and further explained in Section 7.

6. Sorting Equations for Short Access Distance

One part of translating an acausal equation-based model into simulation code involves sorting the equations by data dependency order. This is done using Tarjan's algorithm, which also finds any strongly connected components in the system graph, *i.e.*, a group of equations that must be solved simultaneously. This phase of the compilation corresponds to the Analyzer block in Figure 1.

	$Proc_1$		$Proc_2$	
	V_1	V_2	V_3	v_4
eq_1	1			
eq_2	1	1		
eq_3	1		1	
eq_4			1	1

Figure 3. Incidence matrix in block lower triangular form. Occurrences in the grey area mean a dependency between processor 1 and processor 2.

One way to represent the information about dependencies between equations and equation sorting is through an incidence matrix. In such a matrix there is a row for each equation and a column for each unknown variable in the system. Each position in the matrix is marked if the variable corresponding to the column appears in the equation corresponding to the row.

In Figure 3 we see a small example of such a matrix after equation sorting has been carried out. In this form, we can read out which variable is solved for in which equation by looking at the diagonal. We can also see that, if we calculate the values for the variables in the order they appear in the columns (in this case, v_1, v_2, v_3, v_4), all dependent variables will be calculated before they are needed in another equation.

We assign a sequence number to each variable, or set of variables in case of a strongly connected component, and use this to help the scheduler assign tasks that communicate much within the same processor. When the task graph is generated, each task is marked with sequence number of the variable it calculates. When a system with n variables is to be scheduled onto p processors, tasks marked 1 through n/p are assigned to the first processor and so on. This guarantees that within one evaluation of the right hand side all dependencies that cross process boundaries point in the same direction. See Section 7 for more details on scheduling

We define the *access distance* of the list of equations to be the maximum distance between two equations within the sorted list of equations where one uses the result calculated by the other. If this access distance is small, the risk of a dependency crossing the process boundary is also small.

Data dependencies also exist between simulation time steps. The computed state variable values from one time step are input to the next time step. These dependencies, unlike those within a single time step, normally go in both directions. The forward inter-step dependences are not a problem since higher ranked processors always will be slightly behind in the calculation. It is however important that the backward dependencies also have a short access distance. Fortunately, models with short access distance of the first kind tend to have short access distance between time steps as well. Figure 4 shows the data flow between processors. As can be seen in the figure, if the dependencies would stretch over more than one processor boundary the pipeline would be stretched out as well.

Even though Tarjan's algorithm assures that the equations are evaluated in a correct order, we cannot be sure that there is not a different ordering where the access distance is smaller. If, for example, two parts of the system are largely independent, they can become interleaved in the sequence of equations, making the access distance unnecessarily large. Therefore we apply an extra sorting step after Tarjan's algorithm which moves equations with direct dependencies closer together. This reduces the risk of two tasks with a direct dependency getting assigned to different processors. What we in effect are doing is to try to make the incidence matrix as narrow banded as possible by permuting the rows and columns, moving occurrences in the lower left part of the matrix closer to the diagonal, while keeping the block lower triangular form. See [6] for an explanation of this algorithm.

Assigning tasks to processors is the same thing as dividing the sorted incidence matrix into groups of adjacent columns and assigning each task involved in an equation where a variable belonging to a specific group is solved, to the processor corresponding to that group. The presence of a non-zero element in the grey area of the matrix in Figure 3 corresponds to a dependency between processors, so the fewer columns each process gets assigned the narrower the band of the matrix must be to avoid dependencies. This leads to the conclusion that there are three variables that influence how successful the scheduling is, namely;

- The number of processors
- The width of the band of the matrix.
- The size of the matrix

If the width of the band in the matrix is kept the same, more processors can be utilized if the model gets larger. If the band gets wider but the problem size, i.e., the size of the matrix stays the same, fewer processors can be utilized. As a consequence we can note that for models that originate from a discretization, like the model used in the measurements in Section 8, the finer discretization used, the more processors can be utilized.

7. Scheduling

In this section, we describe the scheduling process. We want all communication occurring inside the same time step to be one-way only, from processors with lower rank to processors with higher rank.

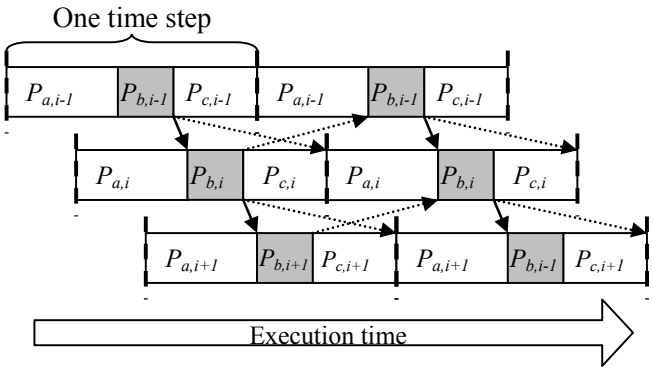


Figure 4. Pipelined scheduling of task sets $P_{a,i}$, $P_{b,i}$, and $P_{c,i}$ on three threads. Solid arrows represent communication of results needed within the same time step and dotted arrows represent communication of results needed in the next time step.

To achieve this, we make use of information stored with each task telling us from which equation it originates from, and thus, which variable's evaluation it is part of. We do this by assigning the tasks to the processors in the order obtained after the sorting step described in Section 6.

Tasks with variable number 1 through n_1 are scheduled to the first processor, n_1+1 through n_2 to the second and so on. The values of n_i are chosen so that they are always a variable number representing a state variable.

The second goal of the scheduling algorithm is to generate results that are needed for the next time step by other processors as early as possible so that a lower ranked processor can start the processing the next time step as early as possible.

To achieve this we collect all tasks that are involved in calculations of results that are needed by other processors and which also themselves depend on results of other processors into one set which we keep as small as possible.

In order to explain how the tasks are sorted within one processor we first introduce some definitions:

$dep(t_1,t_2)$	If t_1 and t_2 are two tasks in the set of all tasks P, the relation $dep(t_1,t_2)$ holds iff there is a path from t_1 to t_2 in the task graph, i.e., t_1 is an ancestor of t_2.
P	The set of all tasks
P_i	The set of tasks assigned to processor i.
$P_{a,i}$, $P_{b,i}$, $P_{c,i}$	Subsets of P_i, such that for all tasks a in $P_{a,i}$ and b in $P_{b,i}$ **not** $dep(b,a)$ holds and for all tasks b in $P_{b,i}$ and c in $P_{c,i}$ **not** $dep(c,b)$ holds.

These sets are populated according the following formulas:

$$t \in P_{b,i} \text{ iff } t \in P_i \wedge \exists u,v(u \in P \wedge u \notin P_i \wedge dep(u,t) \wedge v \in P \wedge v \notin P_i \wedge dep(t,v))$$

$$t \in P_{a,i} \text{ iff } t \in P_i \wedge t \notin P_{b,i} \wedge \exists u((u \in P_{b,i} \vee u \notin P_i) \wedge dep(t,u))$$

$$t \in P_{c,i} \text{ iff } t \in P_i \wedge t \notin (P_{a,i} \cup P_{b,i})$$

The formulas above should be interpreted as follows: A task t assigned to processor i belongs to the set $P_{b,i}$ if these two statements hold:

- A result calculated by a processor other than i is used somewhere on a path leading to t in the task graph.
- There is a task assigned to a processor other than i, on a path leading from t.

A task t assigned to processor i belongs to the set $P_{a,i}$ if there is a task assigned to a processor other than i on a path leading from t, or in $P_{b,i}$.

All other tasks assigned to processor i belong to $P_{c,i}$. The dependencies between these sets are also illustrated in Figure 4.

Let $t_{b,i}$ denote the execution time of the tasks in $P_{b,i}$, let $t_{ac,i}$ denote the sum of the execution times of $P_{a,i}$, and $P_{c,i}$. Furthermore let t_l denote the communication latency.

To keep waiting times as short as possible the relation $t_{b,i} + 2\ t_l < t_{ac,i}$ should hold in each simulation step. This requires the work load to be well balanced between the processors and the computation cost for each task to be reasonably constant.

For many problems, $t_{b,i}$ remains relatively constant when changing the number of processors, whereas $t_{ac,i}$ decreases with the number of processors. This means that, in theory, it is possible to calculate in advance how many processors can be utilized without introducing waiting time.

If the computation cost for each task can be calculated in advance, the load balancing can be carried out in the first step of the scheduling by assigning state variables one by one to the first processor until $1/n$ of the total work has been allocated and then start assigning to the next processor. If, however, the computation cost cannot be accurately estimated, an initial guess can be generated and simulated for a short period of time, to collect measurements so that the processor assignment can be refined in a second attempt.

8. Hardware Platforms and Measurements

The backend of the OpenModelica compiler has been modified so that it can generate parallel code (using the approach just described) for two different architectures: Intel Xeon and SGI Altix 3700 Bx2. A future goal is to be able to generate code for the Cell BE architecture as well. As a first step we have made a small test implementation consisting of a porting of parallel code (generated from the OpenModelica compiler) to the Cell BE architecture.

8.1. Test Model

In order to evaluate the gained speedup we have used a model of a flexible shaft using a one-dimensional discretization scheme. The shaft is modeled using a series of n rotational spring-damper components connected in a sequence. In these tests we use a shaft consisting of 100 spring-damper elements connected together. The Modelica source code for the test model follows below. The Model is built using components from the Modelica standard library apart from the `SpringDamperNL` model which is a modified version of the `SpringDamper` that appears in the standard library. The modified version uses a non-linear spring-damper model, which is computationally harder and increases the ratio between computation and communication.

```
model ShaftElement
```



```
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.TwoFlanges;
  Rotational.Inertia  inertia1;
  SpringDamperNL  springDamper1(c=5,d=0.11);
equation
  connect(inertia1.flange_b,
    springDamper1.flange_a);
  connect(inertia1.flange_a, flange_a);
  connect(springDamper1.flange_b,flange_b);
end ShaftElement;

model FlexibleShaft
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.TwoFlanges;
  parameter Integer n(min=1) = 3;
  ShaftElement shaft[n];
equation
  for i in 2:n loop
    connect(shaft[i-1].flange_b,
      shaft[i].flange_a);
  end for;
  connect(shaft[1].flange_a,flange_a);
  connect(shaft[n].flange_b,flange_b);
end FlexibleShaft;

model ShaftTest
  FlexibleShaft shaft(n=100);
  Modelica.Mechanics.Rotational.Torque src;
  Modelica.Blocks.Sources.Step c;
equation
  connect(shaft.flange_a,src.flange_b);
  connect(c.y,src.tau);
end ShaftTest;
```

If the number of elements used in the discretization is increased, the width of the banded incidence matrix still remains the same. Thus, if we choose to increase the accuracy of the model by increasing the number of elements, we can use more processors and keep the simulation time approximately the same.

The same model has been used in the evaluation of the task merging approach [10], which makes it possible to compare the results of this work to what was previously achieved. The previous results using task merging are shown in Figure 5.

The generated code is divided into regions according to the task sets described in Section 7 and illustrated in Figure 4, instrumented such that a time stamp is stored each time execution moves into a new region of the generated code. In this way it is easy to analyze how much time each thread spends waiting for the result of another processor. The overhead introduced by this instrumentation is negligible.

8.2. Intel Xeon and SGI Altix 3700 Bx2

8.2.1. Implementation

Two approaches have been investigated: 1) all threads working on the same state variable array, or 2) each thread keeping its own state array letting the runtime environment copy results between threads as needed. We also carried out the tests on two different hardware configurations.

The first configuration consists of a standard 3 GHz PC with a 4 core Intel Xeon processor with Hyper-Threading, which means that the operating system sees eight cores, but two hardware threads share the same execution resources. Hyper-threading enables the processor to quickly switch between two threads if one of them stalls due to a cache miss, branch misprediction, or data dependency. The second configuration used is a 64-processor SGI Altix 3700 Bx2 with Intel Itanium 2 processors running at 1.6 GHz.

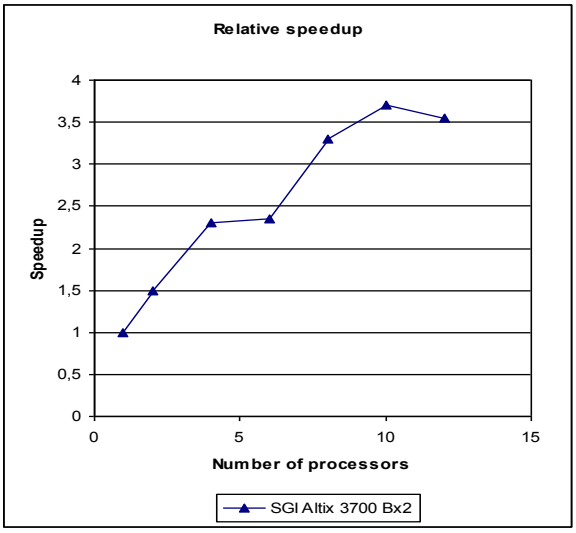


Figure 5. Speedup on the 64 processor SGI machine using the task merging approach on the flexible shaft model. See [9][10] for more details.

8.2.2. Measurements

The first tests were carried out on the Intel Xeon multicore processor using the version where all threads share the same state variable array and the result of running the flexible shaft model on the 8 virtual cores is shown in Table 1. The figures in the table represent mean values over the entire simulation. However, in the raw data we can see that the waiting times vary from almost zero to several milliseconds, indicating that the execution times differ quite a lot between the time steps. This probably has to do with the fact that the cores of the computer are overutilized and some threads are

bound to be preempted by other threads from time to time. Running the same model on only four threads, so that only one hardware thread per core need to be utilized, results in waiting times of about 7 % on all threads. It also becomes a lot easier to balance the work load between the threads since the computation time is more predictable.

Thread	T_{tot} (µs)	T_b (µs)	T_{wait} (µs)	% waiting
1	38.1424	6.19957	4.93	11%
2	38.8396	7.13443	4.23	10%
3	34.1659	8.95761	8.91	21%
4	32.9833	5.90638	10.09	23%
5	34.9385	9.10835	8.14	19%
6	41.887	7.27173	1.19	3%
7	41.5482	9.1998	1.53	4%
8	31.6192	3.47542	11.54	27%

Table 1. Measurements of running the flexible shaft model on eight threads (on Intel Xeon). T_{tot} represents the mean time each thread spends doing useful calculations. T_b represents the mean time each thread spends evaluating tasks in the $P_{b,i}$ set of tasks, i.e., the portion of the code that is on the dependency path between different threads. T_{wait} represents the mean time each thread spends waiting on the result of another thread.

The result indicates that relatively little time is spent on waiting. However, when we compare the execution time to running the same model sequentially on one processor the speedup is only about 2.3. When we, on the other hand, let each thread work on its own memory area and add code to distribute results as needed between the threads instead of doing shared memory accesses, the speedup increases to 4.1 on a test that is identical in all other respects. Thus, the first version using a shared state variable array was abandoned and in the rest of the tests we use a version where each thread keeps its own state variable array and computational results are copied explicitly between dependent threads.

When running the test on the SGI machine, which does not suffer from hardware threads sharing the same core, the test scales well up to eight processors as can bee seen in Figure 6 where a speedup of 6.1 is reached using eight processors. This can be compared to the speedup achieved using the task merging approach investigated in [10] where a maximum speedup of about 3.7 was reached using ten processors on the same hardware.

We also made tests on the SGI configuration using the original linear spring-damper model of the standard library, which has orders of magnitude computationally less expensive tasks. The measured speedup using 4 processors using the linear model is 1.9.

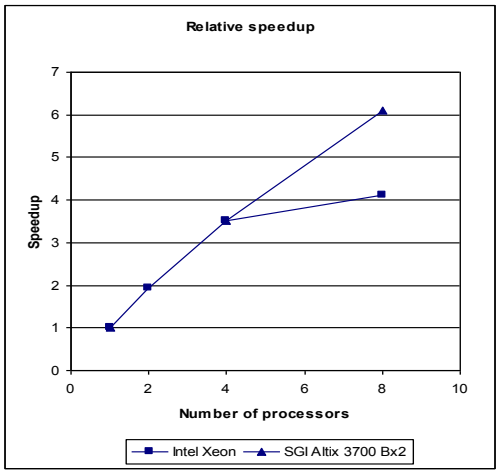


Figure 6. Relative speedup of running the flexible shaft model on the Intel Xeon and the SGI shared memory machines.

8.3. Cell BE

Future work will include the development of a run-time system for the Cell BE processor [14] and a generalization of the current code generator. As a first step, we have manually retargeted the generated parallel C/C++ code (from the OpenModelica compiler) to the Cell BE processor architecture. We took the compiled code for the flexible shaft model mentioned in Section 8.1 (generated for 6 processors) and changed the architecture specific sections to make it work on the Cell BE architecture. Thus the code follows the approach described in Sections 4 to 7.

8.3.1. Implementation

The currently available Cell BE Architecture (CBEA) is a single-chip multiprocessor consisting of one so-called Power Processor Element (PPE) and 8 so-called Synergistic Processor Elements (SPE). The SPEs are optimized for running compute-intensive applications while they are coordinated by the PPE which runs the top level thread and the operating system. The SPEs and PPE do not share on-chip memory. Instead, each SPE has its own, small local on-chip memory for both code and data. DMA transfers are used to transfer data between main memory and the SPEs, and between the different SPEs. As DMA transfers can run asynchronously in parallel with local SPE computation, it should be possible to hide the communication latency during computation.

In the PPE, 6 pthreads are created and loaded with 6 different program handlers pointing to different programs containing different subsets of the equations to be calculated. After creating the threads, the PPE sends out a pointer to a

control block in main memory, with the help of the mailbox facility, to each SPE. Each SPE reads the pointer from its mailbox and uses this pointer to transfer a copy of the control block via DMA to its local store. The control block contains pointers to different buffers in main memory. The SPEs will use these pointers to fetch and store data from main storage and when sending and synchronizing between different SPEs.

After having read the control block from the main store into local store, each SPE reads the init data (for the different vectors X', X, Y and P mentioned in Section 2.2) from a main memory buffer into local store. After this each SPE will iterate N steps, calculating new values, X[t+h], for the state variables (associated with that SPE) in each step. During an iteration step, a SPE might have to send and receive data from neighboring SPEs, corresponding to the arrows in Figure 4. This is done by DMA transfers. After the end of each iteration step (or at the end of some iteration steps, in a periodic manner), data is sent back from the SPEs to the main memory buffer. This data will then be written to a result file by the PPE after all threads have finished.

At the moment, only inter-SPE parallelism and DMA parallelism is utilized. However, in order to exploit the full performance potential of Cell, the SIMD instructions of the SPEs need to be leveraged. This requires vectorization of the generated code, which is an issue of future work.

When scheduling for a small example such as the flexible shaft model, the local on-chip memory is large enough to accommodate the code with associated variables and data. However, for larger examples, either time-consuming overlay of multiple program modules in SPE local store or code and data distribution across a cluster of several Cell processors is needed.

DMA transfers have the advantage that an SPE in some cases can continue to execute while the transfer is going on. For instance, at the end of an iteration step an SPE can initialize a DMA transfer of the resulting values of the state variables in this step to the PPE and then continue on with the next iteration step.

8.3.2. Measurements

Running the whole flexible shaft example 100000 iteration steps on the Cell BE processor (with 6 SPUs as mentioned earlier) took about 31.4 seconds (from start of the PPU main function to end of the PPU main function). The final writing of the result to result files is not included in this measurement. Table 2 shows how much time each thread took to execute in seconds and also how much tim DMA transfer (sending, receiving or waiting for DMA to complete) took for each thread.

Thread	T_{tot} (s)	T_{DMA} (s)	% DMA
1	31.39	2.49	7.9 %
2	31.39	12.28	39.1 %
3	31.39	11.10	35.4%
4	31.39	12.25	39.0 %
5	31.39	11.04	35.2 %
6	31.38	4.39	13.9%

Table 2. Measurements of running the flexible shaft model on six threads (on Cell BE).

From Table 2 we can conclude that thread 1 and 6 do not spend much time on DMA transfers. Thread 2 to 5, however, spent more than a third of the execution time on DMA transfers. To start with, the total execution time of 31.4 seconds is pretty bad. On Intel Xeon the same example took 11.35 seconds (using one core) and on SGI Altix 3700 Bx2 it took 22.59 seconds (using one processor). An improved implementation could most likely decrease the time of DMA transfers. Another issue is the fact that on our Cell BE version double precision calculations takes about 7 times more time than single precision. This will be improved in the next version of the Cell BE. If it would be possible to run the whole flexible shaft example on one SPU it would take about 188.33 – 53.55 ≈ 135 seconds (we can leave out the time for DMA transfers and assume that all data is in local store). This gives a "relative" speedup of about 4.3 as seen in Figure 7. Since the Cell BE architecture is a heterogenous architecture, it is not so straightforward to define relative speedup.

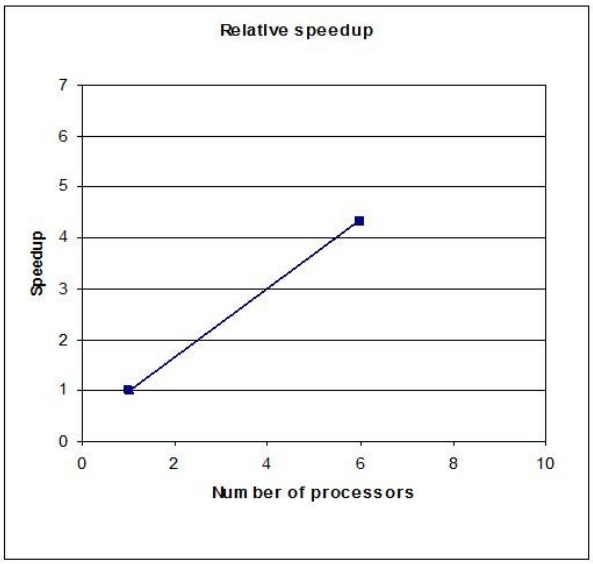


Figure 7. Relative speedup of running the flexible shaft model on Cell BE.

9. Conclusions

We have developed a method and prototype implementation of automatic parallelization of simulation code generated when compiling models in a high level equation-based object oriented modeling language. By using software pipelining, the waiting time each processor experiences can be kept at a low level. Good speedup was achieved on the tested model with 8 processors. Relatively good speedup was achieved on the Cell BE architecture even though more work needs to been done.

If the computation cost can be estimated and communication latency is known, it is possible to predict how many processors can be utilized without introducing waiting times in the computation pipeline.

10. Future work

Tests should be carried out on different simulation problems to see if the results are general or if there are large differences depending on the problem.

A run-time system for the Cell BE processor is also planned. A run-time system for the Cell BE processor has been developed for a related effort on NestStep [4]. Experience from that effort and some code will be reused for the new run-time system. Moreover, the recently developed generative skeleton programming library BlockLib [16] for Cell BE may be useful in future work, as it contains convenient support for efficiently utilizing the SPE SIMD instructions and automatic double-buffering to overlap DMA with SPE computation.

11. Acknowledgements

This work was supported by Vinnova in the Safe & Secure Modeling and Simulation project.

12. References

[1]Håkan Lundvall. Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, Licentiate Thesis No. 1381, Dept of Computer and Information Science, Linköping University, Linköping, Sweden, Sep. 2008.

[2]Håkan Lundvall, Peter Fritzson. Automatic Parallelization using Pipelining for Equation-Based Simulation Languages. Accepted for the 14th Workshop on Compilers for Parallel Computing (CPC'2009), Zurich, Switzerland, Jan 7-9, 2009.

[3]Christoph Kessler, Peter Fritzson and Mattias Eriksson. NestStepModelica: Mathematical Modeling and Bulk-Synchronous Parallel Simulation. PARA-06 Workshop on state-of-the-art in scientific and parallel computing, Umeå, Sweden, June 18-21, 2006.

[4]Daniel Johansson, Mattias Eriksson, Christoph Kessler. Bulk-synchronous parallel computing on the CELL processor. PARS'07: 21. PARS - Workshop, Hamburg, Germany, May 31-Jun 1, 2007.

[5]Ernst Christen and Kenneth Bakalar. VHDL-AMS – A Hardware Description Language for Analog and Mixed-Signal Applications. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing,46(10):1263–1272, 1999.

[6]Håkan Lundvall, Peter Fritzson. Automatic Parallelization of Object Oriented Models across method and system. 6th Eurosim Congress, Ljubljana, Slovenia, 2007.

[7]Joar Sohl. *A Scalable Run-time System for NestStep on Cluster Supercomputers*. Master thesis LITH-IDA-EX-06/011-SE, IDA, Linköpings universitet, 58183 Linköping, Sweden, March 2006.

[8]Matthias Korch and Thomas Rauber. Optimizing Locality and Scalability of Embedded Runge-Kutta Solvers Using Block-Based Pipelining. *Journal of Parallel and Distributed Computing*, Volume 66 , Issue 3 (March 2006), Pages: 444 – 468.

[9]Peter Aronsson and Peter Fritzson. Automatic Parallelization in OpenModelica. In Proceedings of 5th EUROSIM Congress on Modeling and Simulation, Paris, France. ISBN (CD-ROM) 3-901608-28-1, Sept 2004.

[10]Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Dissertation No. 1022, Dept, Computer and Information Science, Linköping University, Linköping, Sweden.

[11]Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In *Simulation News Europe*, 44/45, December 2005. See also: http://www.openmodelica.org.

[12]Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004. See also www.openmodelica.org and book web page: www.mathcore.com/drModelica

[13]The Modelica Association. *The Modelica Language Specification Version 3.0*, September 2007.

[14]Abraham Arevalo et al, Programming the Cell Broadband Engine™: Architecture Examples and Best Practices, IBM RedBooks, Aug. 2008

[15]Francois E. Cellier, Ernesto Kofman. Continuous System Simulation. Springer. ISBN: 0-387-26102-8. 2005.

[16]Markus Ålind, Mattias Eriksson, Christoph Kessler: BlockLib: A Skeleton Library for Cell Broadband Engine. Proc. ACM 1st Int. Workshop on Multicore Software Engineering (IWMSE'08), Leipzig, Germany, May 2008.

[17]Kaj Nyström and Peter Fritzson. Parallel Simulation with Transmission Lines in Modelica. In *Proceedings of the 5th International Modelica Conference (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

[18]Kristoffer Norling, David Broman, Peter Fritzson, Alexander Siemers, and Dag Fritzson. Secure Distributed Co-Simulation over Wide Area Networks. In *Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS'2007)*, available at http://www.ep.liu.se. Göteborg, Sweden. October 30-31, 2007

Paper session 3: Coherence and consistency

A Scalable Directory Architecture for Distributed Shared Memory Chip Multiprocessors

Huan Fang and Mats Brorsson
KTH School of Information and Communication Technology
{huanf, matsbror}@kth.se

Abstract

Traditional Directory-based cache coherence protocol is far from optimal for large-scale cache coherent shared memory multiprocessors due to the increasing latency to access directories stored in DRAM memory. Instead of keeping directories in main memory, we consider distributing the directory together with L2 cache across all nodes on a Chip Multiprocessor. Each node contains a processing unit, a private L1 cache, a slice of L2 cache, memory controller and a router. Both L2 cache and memories are distributed shared and interleaved by a subset of memory address bits. All nodes are interconnected through a low latency two dimensional Mesh network.

Directory, as a split component as L2 cache, only stores sharing information for blocks while L2 cache only stores data blocks exclusive with L1 cache. Shared L2 cache can increase total effective cache capacity on chip, but also increase the miss latency when data is on a remote node. Being different from Directory Cache structure, our proposal totally removes the directory from memory which saves memory space and reduces access latency. Compared to L2 cache which combines directory information internally, our split L2 cache structure saves over 88% cache space while having achieved similar performance.

1. Introduction

As number of cores on a single chip increases, the focus shift from improving computation capabilities on single core to research on interconnection and communication patterns. There are two main issues that designers should consider for large-scale CMPs. One is how we manage on-chip resources like caches, memory controllers and routers. There is a trade-off between cost and performance with different cache hierarchies and memory systems. The other involves design space for interconnection. Traditional bus-based interconnection cannot satisfy increasing bandwidth requirement for tens or even hundreds of cores connecting to each other. We need a more scalable on-chip network than global broadcast technique for future CMPs.

Scalability, as a property of systems, is highly significant in electronics systems, database, routers, and networking. A system, whose performance improves after adding hardware, proportionally to the capacity added, is said to be a scalable system.

In this work, a novel directory-based protocol is designed and simulated to verify if it is suitably efficient and practical when applied to large situations. That is, a large working set and large number of participating nodes in a distributed shred memory multicore processor. We focus system design on hardware scalability rather than on capacity because it is typically cheaper to add a new node to a system in order to achieve improved performance than to partake in performance tuning to improve the capacity that each node can handle.

With the redesign of cache and directory structure, our protocol achieves similar performance as an existing SCMP protocol which integrating directory together with L2 cache. However, the memory overhead is greatly reduced to 6% - 12% compared to SCMP. It performs visible speedup on scalability research as well. With 16 and 64 nodes involved, our protocol shows even better statistics than SCMP in some cases.

2. Related Work

Since point-to-point interconnection network is commonly used for scalable multiprocessor systems, a cache-coherence protocol that does not use broadcast is necessary to store the locations of all cached copies of shared data. A directory-based protocol consists of a list of these cached locations; each one is a directory entry that corresponds to a memory block. There are three primary categories of directory protocol: full-map

directories, limited directories, and chained directories [4]. In this work, we focused on the performance of our directory structure; therefore the full-mapped directory scheme is used for all protocols concerned through our work.

In [1] and [2], a multilayer directory structure is presented to reduce the size of the directory for large-scale configurations of a multiprocessor while also maintain or even improve the performance. The first-level directory and the shared data cache integrated into the processor enhance performance by saving long access time to slow memories. The compressed directories, which are placed outside the chip, are second-level and third-level directories. The compressed data structure significantly reduces memory overhead compared to traditional simple full-map directory.

Directory caches can be also used to reduce the long latency to memory of L2 misses. The directory information is obtained from a much faster structure (caches on chip) than from memory. For example, in [5] a directory cache located together with the L2 is responsible for most of the coherence messages handlings. The directory that stores directory information for every memory block is kept in memory, which does nothing to guarantee cache coherence but functions as a backup storage for sharing information in directory cache. The state information are copied back to main memory only there is a DC (Directory Cache) replacement, thus greatly reduces accesses to directory in DRAM.

A lightweight directory architecture is introduced in [8] that adds directory information to the L2 caches and removes the directory structure from memory. However, this structure increases cache misses due to premature invalidations when a replacement in L2 arises. To minimize premature invalidations like this, [9] proposes a new L2 cache design by splitting L2 cache into two parts: 1.The Data and Directory Information (DDI) structure that maintains both data and directory information for blocks requested by the local processor. 2. The Only Directory Information (ODI) structure that stores only directory information for the local blocks requested by remote nodes and not being used by the local node. The ODI acts like an on-chip directory cache. When a block is evicted from the DDI structure, the ODI structure is used as a victim cache for the directory information of this block. This avoids premature invalidations as a consequence of replacements. But the invalidations are inevitable if a directory entry is evicted from the ODI structure. The ODI+DDI structure is similar to a cache+directory design only that the DDI adds directory information in on-chip cache. Our proposal is to separate L2 cache and directory thoroughly; the address of directory are mapped to L2 cache according to number of sets of L2. Besides, there is no need of directory information to be stored in main memory. Thus it will greatly reduce memory overhead for large-scale multiprocessor system.

Chip multiprocessors place multi processors (cores) on a single die. There are two basic schemes to manage the on-chip L2 cache in tiled CMPs [6]. 1. Private L2 cache for each local processor, or 2. Distributed L2 caches that form a single high-capacity shared L2 cache. The private scheme has low L2 hit latency, performing well if working set fits in the local cache. But it reduces total effective on-chip capacity since each core must keep a local copy of the data it accesses. The structure is simply similar to shrink traditional multi-chip multiprocessors onto a single chip. The shared scheme manages L2 slices as a shared L2 cache with addresses interleaved across cores, which is an example of a non-uniform cache access (NUCA) design. It reduces L2 miss rates and thus memory accesses for large shared working sets. However, the on chip network latency varies depending on distance and network congestion. Most existing CMP designs shared a banked L2 cache while maintain coherence among all primary caches. [6] presents a shared L2 design by adding additional directory bits to each L2 line in order to track shared copies.

From description of [7], we know that UltraSPARC T1 maintains coherence by shadowing the L1 tags in an L2-cache directory structure. The L2 cache directory preserves the inclusion property – all valid entries in the primary cache should reside in the L2-cache as well.

In this work, we present a distributed shared L2 design that preserve exclusive L1/L2 property. Any data block can reside in L2 cache only when it's not in any L1 caches, otherwise it's removed from L2. The directory information is stored in a distributed directory structure if any node(s) has copies of this block in their primary caches.

3. Parameter Settings

We target our CMP chip design assuming 45nm technology in 2010 and 22nm technology in 2016 [5]. Each CMP design is restricted within 400 mm^2 die area. For a 64-core multiprocessor with 22nm technology, we estimate the cores (including L1 I&D caches per core) would occupy 320 mm^2 and 8MB L2 cache would occupy 48 mm^2 area [3], the

interconnection and I/O interface occupy the remaining area.

In order to study the scalability of many cores chip multiprocessors, we have three configurations varied from 4, 16 to 64 cores. The system is modeled by Simics and Gems based on a default ruby configuration file. Cache size is chosen based on practical processor design and working sets of our benchmark programs. It's sufficient for application like blackscholes, but still too small for large application like canneal. Since our concentration is the influence of directory component and scalability of a reasonable system, other parameters are fixed as stated in table 1.

Table 1: System parameters.

	4-core	16-core	64-core
Processor	1 GHz, 2 IPC		
L1 I & D cache	32 KB, 4-way		
Total shared L2 cache	2 MB,	4 MB,	8MB,
Associativity	4-way,	4-way,	4-way,
Banks	4 banks	16 banks	64 banks
L1/L2/Dir block size	64 Bytes		
Directory Associativity	4-way		
Memory size	4 GB DRAM		
On-chip link latency	1 cycle		
L1 hit latency	1 cycles		
L2 hit latency	2 cycles		
Directory latency	4 cycles		
Memory latency	60 cycles		
Topology	2D Mesh		

4. Approach

4.1. System Architecture

In this section, we present our hardware model generated by Virtutech Simics simulator. We model a serengeti machine with up to 64 processors, 4GB memories in total. The detailed cache and memory system are modeled by GEMS.

4.1.1. Hardware Model. We configure our system as a single chip multiprocessor with 4, 16 or 64 cores, the same number banks of L2 caches and memories. Both L2 cache and memories are distributed shared and interleaved by a subset of the physical address bits. All nodes are interconnected as a Mesh network, providing scalable bandwidth as well as simple hardware implementation. Each node contains a processing unit, private L1 cache, a slice of L2 cache, memory controller and a router. A hardware abstract model with 4 CPU cores is shown in Figure 1.

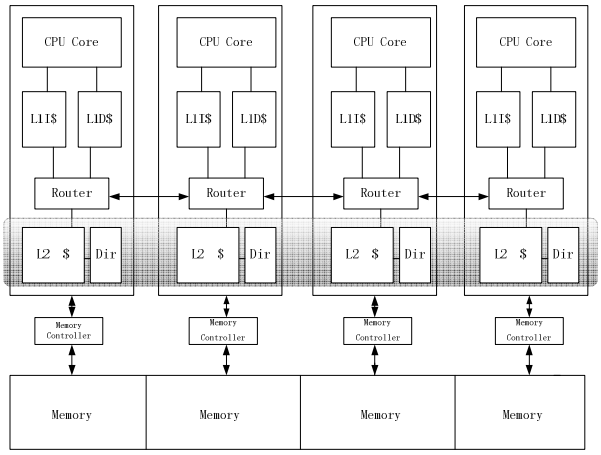


Figure 1: Proposed Architecture.

4.2. Cache Coherence Protocol

4.2.1. Proposed cache organization. In each node, we model a directory of variable size determined by number of nodes in this system. Besides, the number of entries of directory can be set individually as that of the L2 Cache. It is hard to decide the optimal size of directory, which depends on the property of application and size of working set. But we will try to find a threshold that satisfies most applications. Beyond the threshold value, no obvious improvement can be observed. Thus we are able to get a good trade off of performance and cost.

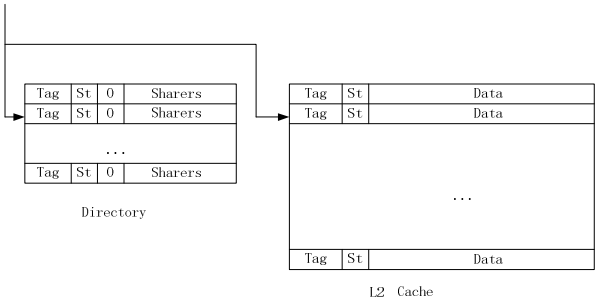


Figure 2: Directory and L2 Cache organization.

With the structure shown in figure 2, the data blocks are stored in L2 cache only in Idle state. Thus sharing information is not necessary for L2 Cache. When there is a request from any L1 cache, the block will be sent to the destination and a directory entry is allocated, storing its state, owner and sharers if any. When a directory replacement is triggered, invalidations to all sharers are required and the clean data is going to be written back to L2 Cache. There is no need to store

anything back to memory for now until a L2 replacement is triggered. This mechanism significantly reduces memory accesses compared to the unified directory + L2 cache structure.

However, the indispensable invalidations increase L1 misses. More coherence messages and on-chip cache-to-cache transfers are generated through the interconnection network. But we believe with high bandwidth provided by network on chip, it's worthwhile to trade some on chip traffic for less off chip traffic. Another explicit and key advantage is the minimized memory overhead for directory structure.

A directory entry consists of four parts: Tag, State, Owner and sharers. The Tag is the ID of a data block as in normal cache. The other bits have same meaning as typical directories. The whole directory is organized as four way set-associative cache. The number of cache blocks mapped to the same set in directory is fixed and determined by address.

The replacement policy used in the protocol is called "Last Recently Used" (LRU) policy. Since we have a four way associative directory, we need to make a replacement when we have four blocks cached in one set and then a fifth one is requested by another processor. In this case we need to invalidate one directory entry. The procedure will be the following: first processor Px requests a block that generates the replacement, the cache controller first stalls Px, then invalidates the block that was not used for the longest time (according to LRU policy), when the invalidation is finished and an empty entry is allocated, Px is given access to the requested block [5].

5. Simulations

5.1. Simulation Metrics

1. Execution time: The Ruby Cycle is our basic metric of simulated time used in the Ruby module of the GEMS simulator. The Ruby module is the basic module for the memory system configuration and interconnection network design. The value of ruby cycles is the count of the number of times the ruby event queue is invoked in the course of simulation. The ruby event queue is invoked every two Simics cycles from the Simics event queue associated with Simics processor 0. Each ruby cycle is one simulated cycle of the memory system analogous to one cycle of a logical memory clock. Ruby cycles are not determined by the number of instructions executed on any processor. The Ruby cycles are the recommended performance metric in GEMS.
2. L1 cache misses: As the name says it represents the misses of L1 cache. It's calculated by dividing request missed by number of requests (Instruction + Data). It's an important metric for cache hierarchy.
3. L2 miss/miss rate: This represents the total misses and miss rate of the L2 cache. It is calculated from the number of requests issued to the L2 and the misses of all banks of L2.
4. L2/Dir replacement: Number of replacements of L2/Directory entries. It's caused by capacity misses and conflict misses.
5. Miss latency average: Average of the L1 miss latency in Ruby cycles. It is measured from the moment a memory request is issued to the moment when the date is retrieved.
6. Memory requests: Number of reads and writes issued to main memory.

5.2. Results and Analysis

Before we start testing with benchmark tools, it's important to know the main improvement of our protocol is significantly reduced overhead compared to traditional architecture. The directory consumes less space while maintaining similar performance. It can be further reduced if combining with other mechanism like compressed directory code. In figure 3, the last column shows SCMP protocol which has 8192 directory entries.

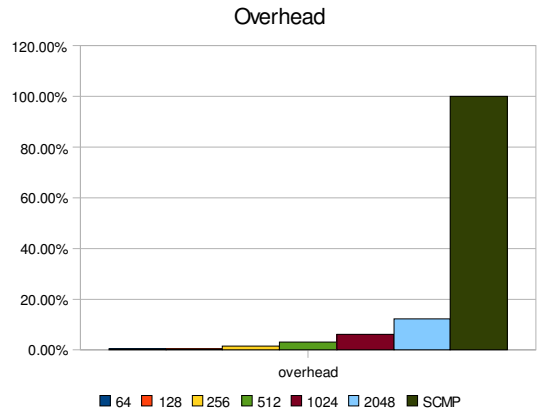


Figure 3: Overhead comparison (directory entries).

5.2.1. Influence of directory size. In order to analyze the impact of directory on the system, we will first vary the directory size from 64 to 2048 entries on a four node Chip multiprocessor. Another protocol with unified directory+L2 cache structure is also simulated for comparison. We call it SCMP for short. All simulations are configured with same cache size: 32KB L1I + 32KB L1D and 512KB L2 cache per core.

Since the SCMP requires same number of directory entries as L2 Cache, the overhead is up to 100%. Reducing directory from 64 to 2048 entries, we get much fewer overhead from 0.39% to 12.5% compared to SCMP, which will save a great amount of die area for multicore processors.

Three PARSEC programs with different characteristics are chosen to represent certain range of applications.

- Blackscholes has least instruction counts and minimum data requests;
- Swaptions has most instruction counts and medium working set.
- Canneal has medium instruction counts but unbounded working set.

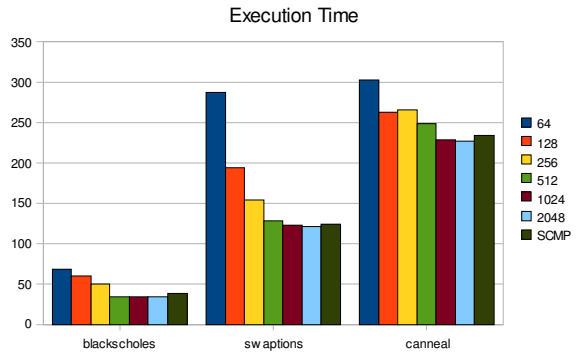


Figure 4: Execution time (million cycles).

As we see from figure 4, with 512 entries for blackscholes, it has almost achieved the best performance. The ruby cycles is improved by 105% compared to 64 entries! However, hardly could we improve performance any more by increasing directory size. It also beats SCMP protocol by 14% improvement.

The 136% improvement for swaptions is even more remarkable. It is also a little better than SCMP Protocol due to less L2 misses and memory references. The best directory size for it is between 512 ~ 1024. Actually 512 entries is enough for most applications.

As to canneal, the influence of directory size is not as significant as the other two. It achieved about 33% improvement ranged from 64 to 2048 entries, only 3% improvement over SCMP. The reason is that the large working set of canneal makes performance constrained by cache capacity to a great extent. As we will see later, the large cache misses limit the system performance. For applications like this, at least 1024 or 2048 directory entries are needed, or ¼ of L2 cache entries.

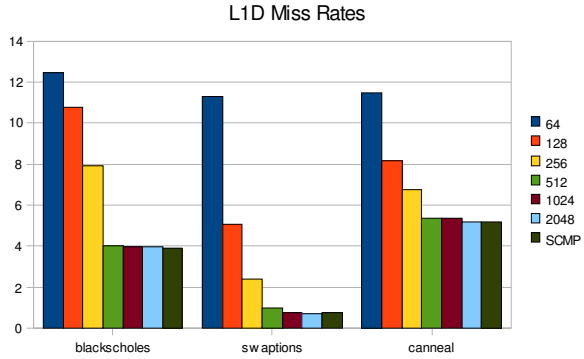


Figure 5: L1 data cache miss rate (%).

The L1 data cache miss rate is calculated as dividing number of L1D misses by data requests. L1 miss is greatly affected by directory size. It again influences L2 requests and on-chip traffics. Therefore, it's an important factor that determines CMP performance.

As seen in figure 5, all programs have a descending L1 miss rate especially for swaptions. However the descending is not that obvious when directory size is beyond 512 entries. We should also notice that even 512 directory entries have already approximated L1 miss rate of SCMP protocol, while the latter has the same directory entries as L2 (8192 in this case).

The variation of directory entry replacement is the most direct outcome of descending directory size. It decreases from tens of millions to tens of thousands while brings outstanding performance change. Basically, designers are supposed to keep these kinds of replacements below 1 million. Figure 6 shows the detail.

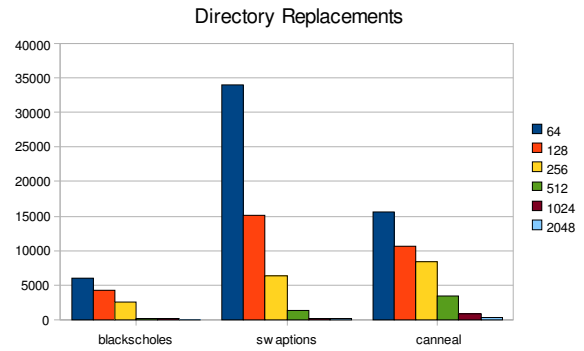


Figure 6: Directory replacements (10^3).

Since we have exclusive L1/L2 cache design, any write backs from L1 will miss in L2. L2 cache acts as a victim cache. No modifications will be made to data in L2 cache. Therefore, we record only read misses for L2, and calculate miss rates by dividing read misses by read requests.

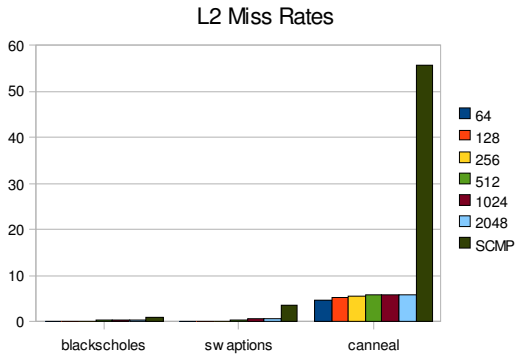


Figure 7: L2 miss rate (%).

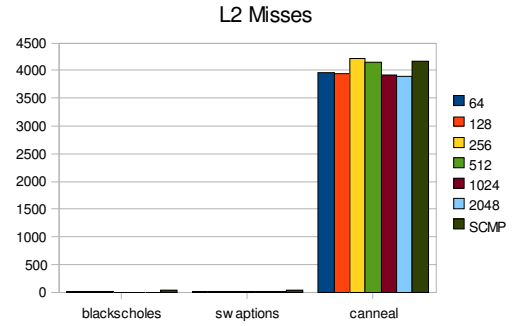


Figure 8: L2 misses (10^3).

L2 misses is mainly determined by L2 cache size and application working set. From figure 8, we can see huge misses (>100x) of canneal compared to blackscholes and swaptions. The working set of canneal can not fit in L2 cache for both structures, thus there is no obvious difference in number of misses. But for blackscholes and swaptions, cache size is a dominant factor. With exclusive caches, our protocol possesses larger effective cache capacity. It reduces a large amount of misses and achieved smaller miss rate.

Figure 7 shows that L2 miss rate of SCMP for canneal is as high as 55%. That is because the total L2 requests is only 10% compared to others.

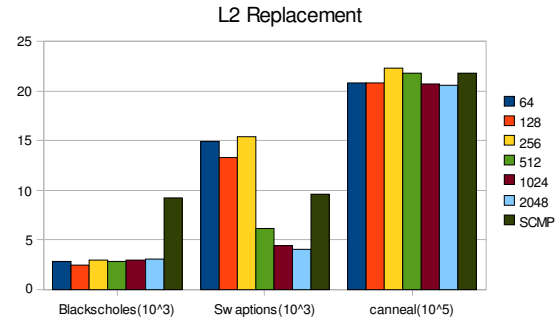


Figure 9: L2 replacements (10^3).

L2 Replacement occurs when the L2 cache is full and another allocation is required. According to LRU policy, a block will be chosen and replaced by a new one. If this block is clean, we can just ignore it and process allocation without pause. Otherwise the data block is to be written back to main memory.

For small applications, replacements of L2 are determined by directory and cache size, the larger the better. But for canneal, the cache capacity is too small to reflect this trend. Only a tiny fraction of working set can fit into cache. Therefore, replacements happen more frequently in this case.

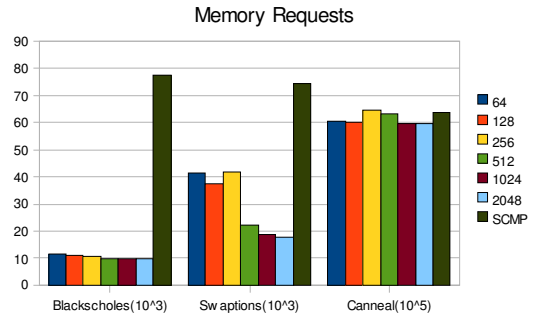


Figure 10: Memory requests.

Memory Requests, to some extent, represents the requirement of memory bandwidth of applications. As we see from figure 10, canneal has over one hundred times memory requests than other two. It corresponds to the description of PARSEC report. With optimized directory structure, our protocol has minimum L2 cache misses and thus much less memory requests than SCMP protocol(not applies to canneal). It immensely relieves pressure on memory bandwidth and network traffic, while also brings smaller latency (figure 11).

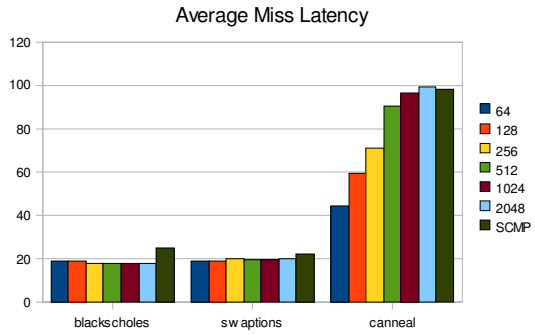


Figure 11: Average miss latency.

5.2.2. Scalability Research. In this section, we are going to study the scalability of our new directory structure and corresponding protocol. A scalable directory-based protocol should work efficiently on

large CMP with up to 64 cores while keep the overhead low. A PARSEC program blackscholes with medium working set is chosen for simulation.

In order to compare with another protocol, we vary the directory size from 512 to 2048 and see whether it can achieve similar performance as the existing SCMP protocol.

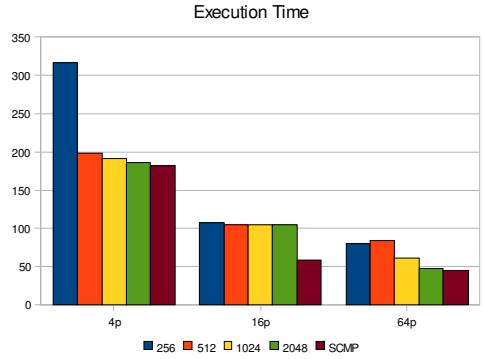


Figure 12: Execution time (million cycles).

The execution time we measured decreases as directory entries double in all cases but 16 core configuration. We can see with 512 entries, our protocol has a performance degression of 9%, 79%, 86% for 4, 16 and 64 cores respectively compared to SCMP. Apparently, 512 entries is far from enough for CMP with tens of CPU cores. Thus we increase directory size up to 2048. The degression is reduced to 2%, 77% and 6%. It's strange that the increasing directory barely has impact on execution time for 16 core configuration. But for 4 and 64 cores, our protocol is approximating SCMP protocol.

We can also check the speedup of program with multiple threads running. The improvement is quite large from 4 to 16 cores. When it comes to 64 cores, the improvement is limited due to longer miss latency and network latency. In addition, lock waiting and load imbalance can also be the reason that constraints speedup.

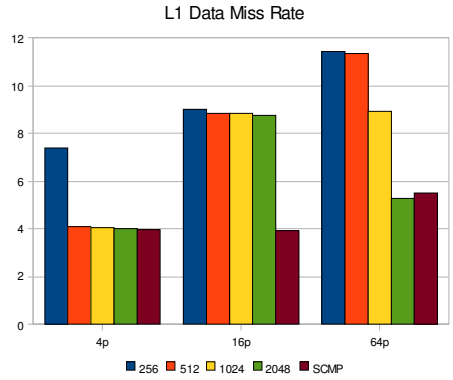


Figure 13: L1 Data cache miss rate (%).

The number of L2 misses increases as we increasing cores within the system, nevertheless the miss rate decrease to 40%-80% because of the larger total L2 cache on chip. In our protocol, the misses consist of cold misses and capacity misses. There is no conflict misses because the data in L2 cache will be read only.

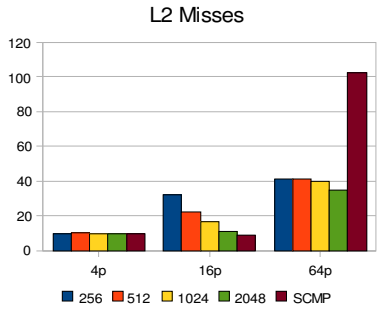


Figure 14: L2 misses (10^3).

We can also see that SCMP has a larger miss rate. The reason is caused by the reduced effective L2 cache size as we discussed before. The interesting thing occurs on 16 core platform, L2 miss rate decreases when directory size increases. It seems the number of directory entries has impact on L2 cache rather than L1 cache. That's why we get almost same execution time statistics.

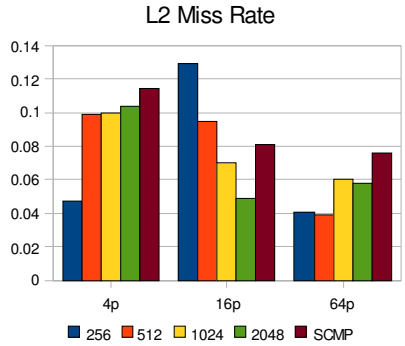


Figure 15: L2 miss rate (%).

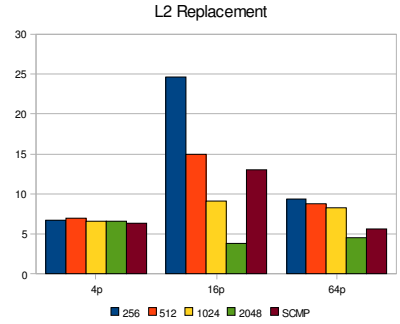


Figure 16: L2 Replacements.

The number of L2 replacement does not vary a lot for 4 cores. In the latter two cases, our protocol has fewer replacements with only 2048 entries. The influence of directory and L2 size becomes dominant for large scale CMP configurations.

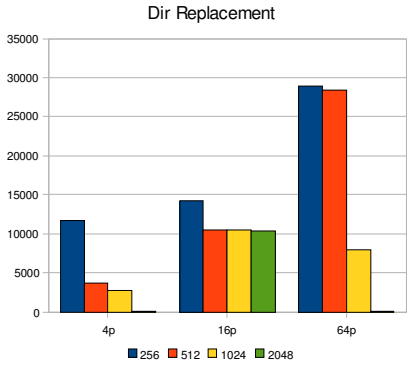


Figure 17: Directory replacements.

However, On 16 core platfrom with 512 to 2048 entries of directory, the number of dir replacements does not change. The reason is that most of the data are mapped to the same location, thus a great many of conflicts happens there. Larger directory size does not improve the situation. We could increase set associativity of directory to avoid the conflicts. The trend seems normal in other two cases.

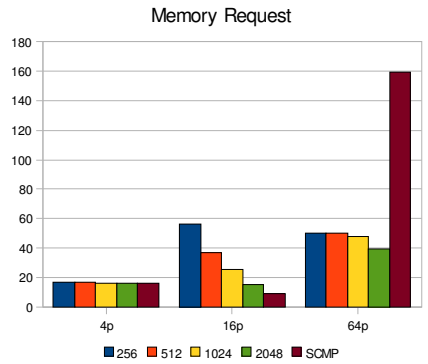


Figure 18: Memory requests.

Compare figure18 with figure14 (L2 misses), the graphs look alike. The memory read requests caused by L2 miss is the dominant faction of total memory requests.

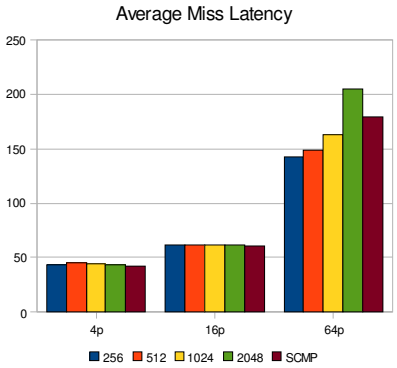


Figure 19: Average miss latency.

The miss latency increases 50% from 4-core to 16-core and 150%-230% from 16-core to 64-core. On a L1 miss, there are up to 3 nodes involved to fulfill the miss: local node, home node and remote node. The 3-way communication aggravates the latency of on-chip communication.

With Garnet network model, we can measure the average network latency in detail. It goes from 14, 22 to 36 in there configurations. For both structure, the latency looks almost the same, which depends on network topology and on-chip link latency. The result we get is acceptable. It will be interesting to investigate other topologies than 2D Mesh.

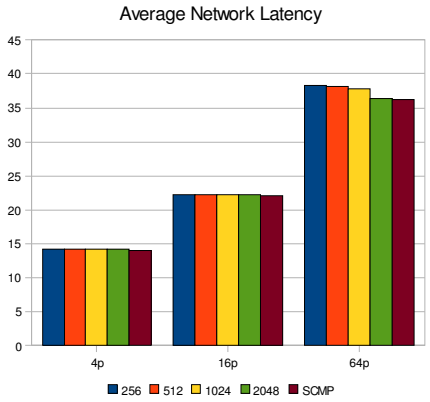


Figure 20: Average network latency.

6. Conclusion and Future Work

In conclusion, our protocol works very well for applications with different characteristics. For large programs like canneal, the improvement might not be so obvious because the bottleneck is cache capacity and memory bandwidth. We believe the improvement will be greater with larger cache size.

Compared to Inclusive L1/L2 cache design with unified directory and cache structure (SCMP), our protocol achieved similar L1 miss rates with only 512 directory entries. Besides, much better L2 miss rates and memory performance make our protocol attractive for future memory hungry applications. Furthermore, greatly reduced memory overhead of directory also reduces hardware cost and leaves more room for other components.

The new directory structure has good scalability on many node processors. The speedup is 1.8x from 4 to 16 cores and 2x from 16 to 64 cores. Although the directory size does not give much influence on execution time for 16 core platform, it's still valuable to study its impact on other aspects.

As you can see, we did not manage to test many programs for scalability research due to time limitation. The simulation result could have been more representative if other benchmark programs involved.

The directory size is reduced in height; it can be further reduced in width by utilizing compressed sharing code. However it implies modifications to GEMS itself since it's not supported natively.

We may also change parameters like network topology defined in ruby to see how the system performance is influenced under every circumstances.

7. References

[1] Acacio, M.E.; Gonzalez, J.; Garcia, J.M.; Duato, J.; An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration. *Parallel and Distributed Systems,* IEEE Transactions on Volume 15, Issue 8, Aug. 2004 Page(s):755 – 768

[2] Acacio, M.E.; Gonzalez, J.; Garcia, J.M.; Duato, J.; A two-level directory architecture for highly scalable cc-NUMA multiprocessors. *Parallel and Distributed Systems,* IEEE Transactions on Volume 16, Issue 1, Jan 2005 Page(s):67 – 79

[3] CACTI 5.3 http://quid.hpl.hp.com:9081/cacti/

[4] Chaiken, D.; Fields, C.; Kurihara, K.; Agarwal, A.; Directory-based cache coherence in large-scale multiprocessors. *Computer* Volume 23, Issue 6, June 1990 Page(s):49 – 58

[5] Enric Herrero Abellanas, Marco Antonio Tirado Godoy, Scalability of a Directory Cache Based Memory Management Protocol in Mesh CMPs. master thesis, Royal Institute of Technology, 2006, Stockholm, Sweden.

[6] The International Technology Roadmap Semiconductors, 2007 Edition, http://www.itrs.net/Links/2007ITRS/ExecSum2007.pdf

[7] Kongetira, P.; Aingaran, K.; Olukotun, K.; Niagara: a 32-way multithreaded Sparc processor. *Micro,* IEEE Volume 25, Issue 2, March-April 2005 Page(s):21 – 29

[8] Alberto Ros, Manuel E. Acacio and José M. García. A Novel Lightweight Directory Architecture for Scalable Shared-Memory Multiprocessors. *Euro-Par 2005 Parallel Processing* pp. 582-591, 2005

[9] Alberto Ros, Manuel E. Acacio and José M. García. An efficient cache design for scalable glueless shared-memory multiprocessors. Conference On Computing Frontiers, *Proceedings of the 3rd conference on Computing frontiers* 2006 Pages: 321 – 330

[10] Zhang, M.; Asanovic, K.; Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. *Computer Architecture,* 2005. ISCA '05. Proceedings. 32nd International Symposium on 4-8 June 2005 Page(s):336 – 345.

State-Space Exploration for Concurrent Algorithms under Weak Memory Orderings (Preliminary Version)

Bengt Jonsson
UPMARC
Department of Information Technology, Uppsala University, Sweden
bengt@it.uu.se

Abstract

Several concurrent implementations of familiar data abstractions such as queues, sets, or maps typically do not follow locking disciplines, and often use lock-free synchronization to gain performance. Since such algorithms are exposed to a weak memory model, they are notoriously hard to get correct, as witnessed by many bugs found in published algorithms We outline a technique for analyzing correctness of concurrent algorithms under weak memory models, in which a model checker is used to search for correctness violations. The algorithm to be analyzed is transformed into a form where statements may be reordered according to a particular weak memory ordering. The transformed algorithm can then be analyzed by a model-checking tool, e.g., by enumerative state exploration. We illustrate the approach on a small example of a queue, which allows an enqueue operation to be concurrent with a dequeue operation, which we analyze with respect to the RMO memory model defined in SPARC v9.

1. Introduction

Shared-memory multiprocessors and multi-core chips are now ubiquitous. Programming such systems remains a huge challenge [19]. To make matters worse, most commonly used multiprocessor architectures use weak memory ordering models (see, e.g., [1]). For example, a processor may reorder loads and stores by the same thread if they target different addresses, or it may buffer stores in a local queue. To avoid exposing the programmer to these complications, programming guidelines recommend to employ a locking discipline to avoid race conditions which expose the particular weak memory model of the target platform. Such programs can be understood as using a sequentially consistent memory semantics [12], and can be reasoned about using standard interleaving semantics.

Concurrency libraries, e.g., the Intel Threading Building Blocks or the `java.util.concurrent` package, support the programmer by providing concurrent implementations of familiar data abstractions such as queues, sets, or maps. Implementations of such libraries typically do not follow locking disciplines, and can use lock-free synchronization for gaining performance (e.g., [13, 18]). Since these algorithms are exposed to a weak memory model, they are notoriously hard to get correct, as witnessed by many bugs found in published algorithms (e.g., [7, 14]). Implementations that use lock-free synchronization require explicit memory ordering fences to function correctly on weak memory models. Fences counteract the ordering relaxations by selectively enforcing memory order between preceding and succeeding instructions. A lack of fences leads to incorrect behavior, whereas an overzealous use of fences impacts performance. Unfortunately, fence placements are rarely published along with the algorithm.

This paper addresses the problem of verifying correctness of, or finding bugs in, concurrent algorithms which do not rely on explicit locking, as found in, e.g., lock-free implementations of common data structures. The absence of locks means that standard race-detection tools (e.g., [9, 16]) are of little use. Existing verification and testing techniques and tools must therefore be adapted to handle also weak memory models.

A large share of all verification, program analysis, and testing algorithms, can very roughly be thought of as performing an exploration of possible sequences of computation steps, starting from some initial state. There is of course a huge variation in how they cover the space of computations, and whether they store intermediate system configurations in order to avoid repeating already performed work. Some such exploration tools include the model checker SPIN [11], the backtracking testing/simulation tool VeriSoft [10], and most testing techniques (e.g., [17]). The goal of our work is to provide techniques to adapt them to handle weak memory orderings. In this paper, we consider the model checker SPIN. More precisely, we present a tech-

nique to adapt models analyzed by SPIN, which are naturally expressed for sequentially consistent memory models in the Promela modeling language, so that they also represent all possible computations under a weak ordering.

Some Related Work The research performed on the problem of analyzing concurrent algorithms correct under weak memory models is still limited. The work on CheckFence by Burckhardt and Alur [4] use a bounded model checkin approach rather than state-space exploration: they encode possible computations by a constraint system, and use a SAT solver to search for correctness violations. The work closest to ours is that by Park and Dill [15], who have developed an operational encoding of a shared memory with weak ordering constraints, in particular the RMO model used in SPARC v9, and used it to analyze simple synchronization examples from the SPARC architecture manual, using the model checker Murφ [6]. Their work only reports application to very small examples, our aim is to make a more efficient operational representation of the weak memory model, and to be able to analyze more complicated algorithms, such as, e.g., those considered by Burckhardt and Alur [4]. Some specific weak memory ordering has also been considered in program analysis work [8]. Burckhardt and Musuvathi [5] develops a run-time monitoring tool which checks whether concurrent executions are sequentially consistent, by maintaining vector clocks.

2. Representing Weak Memory Models

In this section, we describe the principles for representing the memory model in this work. Abstractly, a memory model specifies how the program operations "see" the effects of other program operations through the memory system. The interesting part here is how load operations see store operations. More specifically, an execution consists of a set of load and store operations (plus memory barriers, to be explained later), which affect the "state" of the main memory. Each load sees the value of some store operation (or the initial value) to the same location. The hard part is to describe in a concise way which store operations can be seen.

We follow Burckhardt [3] (who in his turn follows previous work), and use

- a partial order $\prec$, called the program order, which is a total order on all operations of the same thread, and which does not order two operations of different threads,
- a total order $<_M$, called the memory order, which intuitively models the order in which operations reach the "main memory".

The fact that $<_M$ is a total order implies that we are aiming at modeling memory models with a global store order, i.e., such that the stores of a thread are seen in the same order by all other threads. Load operations of the thread that performs the store may see it earlier than other threads through the mechanism of store-load forwarding.

The orderings $\prec$ and $<_M$ are related by four axioms. For a load operation l, let *seed*(l) be the store operations which stores the value that l loads. Let $S(l)$ be the set of store operations s which access the same address as l, such that either $s <_M l$ or $s \prec l$. The axioms are

(A1) whenever x and y are operations to the same address, y is a store, and $x \prec y$, then $x <_M y$,

(A2) *seed*$(l) \in S(l)$ for all loads l,

(A3) *seed*(l) is the maximal element wrp. to $<_M$ in $S(l)$,

(A4) whenever $x \prec f \prec y$ for a fence operation f, and x and y match the type of the fence f (e.g., if f is a load-load fence, then x and y should both be load operations), then $x <_M y$.

We shall in particular consider the RMO memory model, defined by SPARC v9, which is also used by Park and Dill [15], which is nice because it preserves single-thread semantics. This is done by defining a dependency order, which constrains the order between data dependent operations of the same thread. For operations x and y of the same thread where x is a load, we say that $x <_d y$ if y loads a data register that is written by x, or if some control branch instruction between x and y is data dependent on x. Add the axiom

(A5) whenever $x \prec y$ and $x <_d y$, then $x <_M y$.

Roughly speaking, the RMO ordering differs in two respects from the natural sequential consistency model. First, operations of one thread may be reordered, but respecting fences and data dependencies. Second, while the global memory order is a merge of the (possibly reordered) local orderings as in sequential consistency, a a load sees the latest store to the same location in the same thread, if it is later wrp. to $<_M$ than the latest preceding store in memory order.

Finally, let us consider locks. In the examples we have locks which are updated by `lock` and `unlock` operations. In this work, we assume that lock operations are atomic, and that

- For the `lock` operation, a fence is inserted to make sure that the `lock` operation precedes all the following instructions of the thread in memory order.
- For the `unlock` operation, a fence is inserted to make sure that the `unlock` operation succeeds the preceding instructions of the thread in memory order.

3. Illustration of Technique

Our ambition is to consider examples, such as those taken from the thesis of Sebastian Burckhardt [3]. In this section, we use one of them, a two-lock queue, to illustrate how the technique presented in this paper, should work. The code for the queue, in C syntax, is the following, taken literally from [3].

```
 1 #include "lsl_protos.h"
 2
 3 /* ---- data types ---- */
 4
 5 typedef int value_t;
 6
 7 typedef struct node {
 8   struct node *next;
 9   value_t value;
10 } node_t;
11
12 typedef struct queue {
13   node_t *head;
14   node_t *tail;
15   lsl_lock_t headlock;
16   lsl_lock_t taillock;
17 } queue_t;
18
19 /* ---- operations ---- */
20
21 void init_queue(queue_t *queue)
22 {
23   node_t *dummy =
       lsl_malloc(sizeof(node_t));
24   dummy->next = 0;
25   dummy->value = 0;
26   queue->head = dummy;
27   queue->tail = dummy;
28   lsl_initlock(&queue->headlock);
29   lsl_initlock(&queue->taillock);
30 }
31
32 void enqueue(queue_t *queue, value_t val)
33 {
34   node_t *node = lsl_malloc(sizeof(node_t));
35   node->value = val;
36   node->next = 0;
37   lsl_lock(&queue->taillock);
38   lsl_fence("store-store");
39   queue->tail->next = node;
40   queue->tail = node;
41   lsl_unlock(&queue->taillock);
42 }
43
44 boolean_t dequeue
          (queue_t *queue, value_t *retvalue)
45 {
46   node_t *node;
47   node_t *new_head;
48   lsl_lock(&queue->headlock);
49   node = queue->head;
50   new_head = node->next;
51   if (new_head == 0) {
52     lsl_unlock(&queue->headlock);
53     return false;
54   }
55   lsl_fence("data-dependent-loads");
56   *retvalue = new_head->value;
57   queue->head = new_head;
58   lsl_unlock(&queue->headlock);
59   lsl_free(node);
60   return true;
61 }
```

The prefix `lsl` on some operations (for memory management and lock operations) means that they refere to particular definitions of these operations used in [3].

Generating an Analyzable Program In order to see which sequences of loads and stores are in principle generated by these operations, we transform the description into "high-level machine instructions", which are on the same level of abstraction as the above C pseudocode, but obeys the restriction that each statement induces at most one store or load operation. A store operation is typically of the form `*p = v` for some address `p` and value `v`. We allow both `p` and `v` to be locally computable expressions. Analogously, a load operation has the form `r = *p` for some local variable `r` (sometimes called register), and address `p`. The first transformation typically preserves most of the description, but breaks up statements that involve more than one store or load. In order to introduce offset calculations more explicitly, for a field `f` in a structure `struct`, we introduce `[f]` to denote the offset induced by `f`. Thus, if `structp` points to `struct`, then `structp + [f]` points to the field `f` in `struct`.

Let us first consider the `init_queue` operation. We transform the code into

```
   void init_queue(queue)
   {
 1 node_t *dummy =
         lsl-malloc(sizeof(node_t));
 2 *(dummy + [next]) = 0;
 3 *(dummy + [value]) = 0;
 4 *(queue + [head]) = dummy;
 5 *(queue + [tail]) = dummy;
 6 lsl-initlock(queue + [headlock]);
 7 lsl-initlock(queue + [taillock]);
   return
   }
```

This is essentially the same as before. In order to infer which are possible orderings between statements, we should

find the data-dependencies between statements. The only ones in this function are that line 1 must precede lines 2, 3, 4, and 5, through the dependence on `dummy`.

Next we consider the `enqueue` operation. At first, we ignore the `fence` instruction at line 38. We transform the code as follows (e.g., breaking the statement at line 39 into two: one load and one store).

```
    void enqueue(queue, val)
    {
 1  node = lsl-malloc();
 2  *(node + [value]) = val;
 3  *(node + [next]) = 0;
 4  lsl-lock(queue + [taillock]);
 5  queuetail = *(queue + [tail]);
 6  *(queuetail + [next]) = node;
 7  *(queue + [tail]) = node;
 8  lsl-unlock(queue + [taillock]);
 9  return
    }
```

Our next job is to see which orderings in the program order are preserved in the RMO model. We see that data dependencies arise as follows:

$$1 <_d 2 \quad 1 <_d 3 \quad 1 <_d 6 \quad 1 <_d 7 \quad 5 <_d 6$$

It remains to understand the ordering constraints imposed by the lock operation. These ensure that instruction 4 precede all following instructions, and that 8 succeed all preceding instructions. In total, we arrive at the following dependencies:

$$1 <_d 2 <_d 8 \quad 1 <_d 3 <_d 8 \quad 1 <_d 6 <_d 8$$
$$1 <_d 7 <_d 8 \quad 4 <_d 5 <_d 6 <_d 8 \quad 4 <_d 7 <_d 8$$

We can summarize these dependencies in the following diagram.

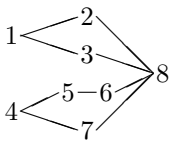


Figure 1. Dependencies in the procedure enqueue

We finally consider the function `dequeue`. A condensed pseudo-code is as follows

```
    void dequeue(queue, retvalue)
    {
 1  lsl-lock(queue + [headlock]);
 2  node = *(queue + [head]);
 3  new_head = *(node + [next]);
 4  if (new_head == 0) {
 5      lsl-unlock(queue + [headlock]);
 6      return(0);
    }
 7  tmp = *(new_head + [value]);
 8  *retvalue = tmp:
 9  *(queue + [head]) = new_head;
10  lsl-unlock(queue + [headlock]);
11  lsl-free(node);
12  return(1);
    }
```

Here, there are more data dependencies.

$$1 <_d 2 <_d 3 <_d 4 <_d 5$$
$$4 <_d 7 <_d 8 <_d 10 \quad 4 <_d 9 <_d 10 \quad 3 <_d 11$$

Lines 6 and 12 should be the last ones

Generating a Promela Model In order to use the SPIN model checker to analyze the queue implementation, we must produce a Promela Model, which executes the statements of the transformed program in any possible order consistent with the ordering. To to this, we must consider the following issues.

- Promela does not support dynamid heap data structures. Instead, we model, e.g., the `queue` structure as just a structure, and the nodes of type `node_t` by an array.
- Promela has only a few standard control constructs, therefore we should find an idiom for allowing all executions that are linearizations of a partial order. We can do this by a loop, which in each iteration checks whether the appropriate preceding statements have been executed in order to see whether some instruction is enabled. This scheme needs an array of flags to record which statements have already been executed.

A possible Promela model of the above queue for a particular test case is shown in Appendix A.

4. Experiments

We have so far only considered the example queue described in Section 3, to obtain some illustrative example. We ran exhaustive analyses using several different test harnesses that first perform an initialization using `init_queue`, and thereafter starts a number of threads, each of which performs a sequence of `enqueue` or `dequeue` operations. After this, we check that the sequence of values returned by the `dequeue` operation is

consistent with a normal sequentially consistent execution of these operations.

We denote test harnesses in a condensed notation (following [3]), using a sequence of e (for `enqueue`) and d (for `dequeue`) in each thread, and separating threads by |. For example, the test (ee | dd) has two threads, one with two `enqueue` operations, and one with two `dequeue` operations.

We first performed a simple test (e | d), which found a shortest counterexample in a few seconds, generating about 900 states. The problem is the obvious one, that the initialization of the new node in `enqueue` at line 2 can be delayed past the dequeueing of the same node, so that the dequeue operation read an uninitialized `value` field. This problem can be remedied by a store-store-fence between lines 3 and line 6 of `enqueue`, e.g., after the `lock` operation, as in the C pseudocode (line 38). This implies that line 6 can be completed only after lines 2 and 3. We modified the promela model accordingly, and reran the test, and the number of reachable states decreased to 250 with no violation of sequentially consistent semantics. The Promela model for this experiment is given in the appendix.

We thereafter subjected the model to the two largest tests of [3], namely (eeeee | ddddd), and (e | e | e | e | d | d). The first test completed by SPIN in less than one second, generating about $100,000$ states. The second test completed after 260 seconds, using state compression and between 1GB and 2GB of memory, generating a state space of $28,000,000$ states. Out of curiosity, we tried different values for the number of operations in the first test, and were able to make SPIN analyze two threads, each with 10 operations, in 143 seconds, generating around $37,000,000$ states. It seems that SPIN has problems handling a large number of threads, due to the many possible interleavings It seems that work on optimization is needed to make the approach scale to a larger number of threads.

5 Conclusions

We presented a technique for analyzing correctness of concurrent algorithms, under weak memory models. The algorithm to be analyzed is transformed into a partial ordering form, which satisfies exactly the ordering constraints imposed by the memory model under consideration. The transformed algorithm can then be analyzed by a model checking tool, such as SPIN.

We implemented the approach in the context of the SPIN model checker [11], by developing a transformation to Promela models, which follows a certain idiom to model execution under partial order constraints. We illustrated the approach by applying it to an example used in the thesis by Burckhardt [3]. the scalability of the approach by applying it to published synchronization algorithms and concurrent data structures.

We should not make to firm conclusions about this approach from the limited amount of experiments conducted. For a better evaluation, the transformation should be automated; now it is by hand. For the particular example considered, the limitations, in terms scalability, appear comparable to the approach by Burckhardt. In our approach, we were able to perform slightly longer test cases, but on the other hand Burckhardt's approach is automated.

The work closest to ours, by Park and Dill [15], use a similar approach of letting a model checker examine all possible executions that are consistent with the memory model. Their work only reports application to very small examples. We have been able to show that the approach can also handle interesting concurrent algorithms.

An impression from the illustrating example is that increase in the number of interleavings as the number of threads grow will impose limits on the scalability in using a model checker in the way proposed in this paper. We can probably make scalability better by introducing a more generic model of the heap; now there will be many duplications of isomorphic heaps in the state space from our representation as an array. It may also be fruitful to consider approaches which are not so sensitive to this explosion, considering either static program analysis (e.g., as in [8] or parameterized infinite-state model checking (e.g., as in [2].

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. IEEE Computer, 29(12):66–76, 1996.

[2] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, Proc. 12^{th} Int. Conf. on Computer Aided Verification, volume 1855 of Lecture Notes in Computer Science, pages 403–418. Springer Verlag, 2000.

[3] S. Burckhardt. Memory Model Sensitive Analysis of Concurrent Data Types. PhD thesis, Univ. of Pennsylvania, 2007.

[4] S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In PLDI 2007, pages 12–21, 2007.

[5] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In Computer-Aided Verification (CAV), pages 107–120, 2008.

[6] D. Dill. The mur*phi* verification system. In Proc. 8^{th} Int. Conf. on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, pages 390–393. Springer Verlag, 1996.

[7] S. Doherty, D. Detlefs, L. Groves, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. S. Jr. Dcas is not a silver bullet for nonblocking algorithm design. In SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain, pages 216–224, 2004.

[8] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In Proc. TAP 2008, 2nd Int. Conf. Tests and Proofs, Prato, Italy, volume 4966 of Lecture Notes in Computer Science, pages 116–133. Springer Verlag, April 2008.

[9] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. Science of Computer Programming, 71(2):89–109, 2008.

[10] P. Godefroid, B. Hammer, and L. Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 124–133, 1998.

[11] G. Holzmann. The model checker SPIN. IEEE Trans. on Software Engineering, SE-23(5):279–295, May 1997.

[12] L. Lamport. Time, clocks and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, 1978.

[13] M. Michael. Scalable lock-free dynamic memory allocation. In PLDI 2004, pages 35–46, 2004.

[14] M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[15] S. Park and D. Dill. An executable specification and verifier for relaxed memory order. IEEE Trans. on Computers, 48(2):227–235, 1999.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 14(4):391–411, Nov. 1997.

[17] K. Sen. Race directed random testing of concurrent programs. In PLDI 2008, pages 11–21, 2008.

[18] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput., 65(5):609–627, 2005.

[19] H. Sutter and J. Larus. Software and the concurrency revolution. ACM Queue, 3(7):54–62, 2005.

Appendix

In this appendix, we show the Promela model, used to analyze the example of Section 3 for the test case (e|d).

```
#define TRUE   1
#define FALSE  0
#define UNDEF  255

#define IF if ::
#define FI :: else fi

#define FOR(i,l,h)    i = l ; do :: i < h ->
#define ROF(i,l,h)    ; i++ :: i >= h -> break od

#define HEAPSIZE    8
#define INITQUEUESIZE 9
#define ENQUEUESIZE 9
#define DEQUEUESIZE 12
#define RETVALSIZE 2

#define malloc(X)  X = cur ; cur++
#define free(X)    \
        atomic{ next[X] = UNDEF ;  value[X] = UNDEF}

byte   next[HEAPSIZE]; /* model of the heap */
byte   value[HEAPSIZE];
byte   cur = 0;

byte   head = UNDEF;  /* the queue structure */
byte   tail = UNDEF;
bit    headlock = 1;
bit    taillock = 1;

                /* stores output from dequeue */
byte   retval[RETVALSIZE];

byte   i = 0 ;

proctype initqueue() {
    bit  done[INITQUEUESIZE];
    byte dummy ;
    do
    :: atomic{!done[1] ->
           malloc(dummy) ; done[1] = TRUE}
    :: atomic{!done[2] && done [1] ->
           next[dummy] = UNDEF ; done[2] = TRUE}
    :: atomic{!done[3] && done [1] ->
           value[dummy] = UNDEF ; done[3] = TRUE}
    :: atomic{!done[4] && done [1] ->
           head = dummy ; done[4] = TRUE}
    :: atomic{!done[5] && done [1] ->
           tail = dummy ; done[5] = TRUE}
    :: atomic{!done[6] && done [1] ->
           headlock = 1 ; done[6] = TRUE}
    :: atomic{!done[7] && done [1] ->
           taillock = 1 ; done[7] = TRUE}
    :: atomic{done[1] && done [2] && done [3] &&
              done [4] && done [5] && done [6] &&
              done [7] ->
           break}
    od
}

proctype enqueue(byte val) {
```

```
    bit  done[ENQUEUESIZE];
    byte node, queuetail;
    do
    :: atomic{!done[1] ->
           malloc(node) ; done[1] = TRUE}
    :: atomic{!done[2] && done [1] ->
           value[node] = val ; done[2] = TRUE}
    :: atomic{!done[3] && done [1] ->
           next[node] = UNDEF ; done[3] = TRUE}
    :: atomic{!done[4] && taillock == 1 ->
           taillock = 0 ; done[4] = TRUE}
    :: atomic{!done[5] && done[4] ->
           queuetail = tail ; done[5] = TRUE}
    :: atomic{!done[6] && done[2] &&
              done[3] && done[5] ->
           next[queuetail] = node ; done[6] = TRUE}
    :: atomic{!done[7] && done[2] &&
              done[3] && done[4] ->
           tail = node ; done[7] = TRUE}
    :: atomic{!done[8] && done[2] && done[3] &&
              done[6] && done[7] ->
           taillock = 1 ; done[8] = TRUE ; break}
    od
}

proctype dequeue(byte rv) {
    bit  done[DEQUEUESIZE];
    byte node, new_head, tmp;

    atomic{headlock == 1 -> headlock = 0};
    node = head;
    new_head = next[node];
    if
    :: atomic{ new_head == UNDEF ->
        headlock = 1 ; retval[rv] = 0}
    :: new_head != UNDEF ->
        do
        :: atomic{!done[7] ->
               tmp = value[new_head] ; done[7] = TRUE}
        :: atomic{!done[8] && done [7] ->
               retval[rv] = tmp ; done[8] = TRUE}
        :: atomic{!done[9] ->
               head = new_head ; done[9] = TRUE}
        :: atomic{!done[10] && done [8] && done [9] ->
               headlock = 1 ; done[10] = TRUE}
        :: atomic{!done[11] && done [10] ->
               free(node) ; done[11] = TRUE ; break}
        od
    fi
}

init{
    atomic{FOR(i,0,HEAPSIZE)
              next[i] = UNDEF ; value[i] = UNDEF
           ROF(i,0,HEAPSIZE)
    } ;
    run initqueue();
    timeout -> atomic{run enqueue(4) ; run dequeue(0)} ;
    timeout -> assert(retval[0] == 0 || retval[0] == 4)
}
```

Model Checking Race-Freeness

Parosh Aziz Abdulla
Uppsala University, Sweden
<parosh@it.uu.se>

Frédéric Haziza
Uppsala University, Sweden
<daz@it.uu.se>

Mats Kindahl
Sun Microsystems,
Database Technology Group
<mats@sun.com>

Abstract

With the introduction of highly concurrent systems in standard desktop computers, ensuring correctness of industrial-size concurrent programs is becoming increasingly important. One of the most important standards in use for developing multi-threaded programs is the POSIX Threads standard, commonly known as PThreads*. Of particular importance, the analysis of industrial code should, as far as possible, be automatic and not require annotations or other forms of specifications of the code.*

Model checking has been one of the most successful approaches to program verification during the last two decades. The size and complexity of applications which can be handled have increased rapidly through integration with symbolic techniques. These methods are designed to work on finite (but large) state spaces. This framework fails to deal with several essential aspects of behaviours for multi-threaded programs: there is no bound a priori *on the number of threads which may arise in a given run of the system; each thread manipulates local variables which often range over unbounded domains; and the system has a dynamic structure in the sense that threads can be created and killed throughout execution of the system. In this paper we concentrate on checking a particular class of properties for concurrent programs, namely* safety properties*. In particular, we focus on* race-freeness*, that is, the absence of race conditions (also known as data races) in shared-variable pthreaded programs.*

We will follow a particular methodology which we have earlier developed for model checking general classes of infinite-state systems [1, 3, 6, 8, 9] and apply a symbolic backward reachability analysis to verify the safety property. Since we construct a model as an over-approximation of the original program, proving the safety property in the model implies that the property also holds in the original system. Surprisingly, it leads to a quite efficient analysis which can be carried out fully automatically.

1. Introduction

The behaviours of concurrent (or multi-threaded) programs are highly nontrivial and hard to predict. It is important to develop rigorous methods to verify their correctness. It is now also widely accepted that verification methods should be *automatic*. This would allow engineers to perform verification, without needing to be familiar with the complex constructions and algorithms behind the tools.

In this paper, we will concentrate on a particular approach to verification of concurrent programs, namely that of *model checking* [4, 16]. The aim of model checking is to provide an algorithmic solution to the verification problem. Concurrent programs involve several complex features which often give rise to infinite state spaces. First, there is no bound *a priori* on the number of threads which may arise in a given run of the system. In addition, each thread manipulates local variables which often range over unbounded domains. Furthermore, a system has a dynamic configuration in the sense that threads can be created and terminated throughout execution of the system.

We concentrate on checking a particular class of properties for concurrent programs, namely *safety properties*. Intuitively, a safety property states that "nothing bad will ever occur during the execution of the system". In particular, we focus on *race-freeness*, i.e. the absence of race conditions (also called data races) in shared-variable concurrent programs.

A race condition is a situation in which one process changes a variable which another process has previously read and the other process does not get notified of the change. A process can, for example, write a variable that a second process reads, but the first process continues execution – namely races ahead – and changes the variable again before the second process sees the result of the first change. Another example: when a process checks a variable and takes action based on the content of the variable, it is possible for another process to "sneak in" and change the variable in between the check and the action in such a way that the action is no longer appropriate. Alternatively,

one can define a race condition as the possibility of incorrect results in the presence on unlucky timing in concurrent programs, that is, getting the right answer relies on lucky timing.

Race conditions are of particular interest because they can lead to rather devious bugs. These bugs are extremely hard to track since they are non-deterministic and difficult to reproduce. The kind of errors caused by race condition are very subtle and often manifest themselves in the form of corrupted or incorrect variable data. Unfortunately, it often means that the error will not harm the system immediately, but it will manifest itself when some other code is executed, which relies on the data to be correct. This makes the process of locating the original race condition even more difficult. To avoid data corruption or incorrectness, the programmer uses synchronization techniques to constraint all possible process interleavings to only the desirable ones. Race condition usually unveil incorrectly synchronized program.

Related Work. Existing race checkers fall into three main categories: on-the-fly, ahead-of-time and post-mortem tools. They exhibit different strengths and can perform race detection, while our method focuses at the moment on race-freeness. The ahead-of-time approach encompasses static analysis and compile-time heuristics, while on-the-fly approaches are by nature dynamic. The post-mortem approach is a combination of static and dynamic techniques.

Type-based solutions provide a strong assurance to the programmer, in addition to the familiarity of a compiler-based solution [10]. If a program type-checks, it is guaranteed to be race-free. It is however necessary to examine the source code as a whole in order to draw conclusion about to find relations between the locks and the shared data that they protect. This is often alleviated by requiring annotations from the programmer, as to whether a function has an effect clause or not, such as "the caller must hold lock L". Moreover, this only forces a programming discipline and it can also disallow some race-free programs.

Dynamic tools visit only feasible paths and have an accurate view of the values of the shared data, while static tools must be conservative. They are based on two main approaches: the lockset and the happens-before approaches. The lockset algorithm works on the assumption that shared variables should be protected by an appropriate lock and any failure to comply to that discipline is reported. Lockset tools tend to report many false positives. This technique has been first implemented in Eraser [17]. The happens-before technique [13] watches for any accesses of shared variables that do not have an implied ordering between them. This technique is highly dependant on the actual thread execution ordering, and instrumentation might bias the analysis. Moreover, it only reports a subset of all race conditions (however real ones). These two main approaches are often combined to get the best of both worlds, e.g. MultiRace and RaceTrack [19, 14].

Model checkers, such as Verisoft, Bandera and KISS [12, 7, 15, 5], check concurrent programs with a finite and fixed number of threads. They often require a user supplied abstraction.

A precise way to detect race conditions would be to search the state space (of a *model* of the program) for a state where multiple threads try to access and change the same variable. The technique is sound (i.e. being able to prove the race-freeness of a concurrent program), but more importantly it seems to be much more precise than the other techniques. It indeed detects the race conditions themselves instead of detecting violations of the locking discipline that can be used to prevent race conditions.

Verification Method. We first construct a model for concurrent programs, where each configuration (snapshot) of the system is represented by a petri net. The transitions in the petri net represent the instructions of the program. The configuration of the system changes dynamically during its execution, by firing transitions in the petri net. The system starts running from an *initial configuration* and we characterize a set of *bad configurations* which violate the given safety property (i.e. configurations which should not occur during the execution of the system). Checking the safety property amounts then to performing *reachability analysis*: Is it possible to reach a bad configuration from an initial configuration through a sequence of actions performed by different threads?

We will follow a particular methodology which we have earlier developed for model checking general classes of infinite-state systems [1]. The method is designed to be applied for verification of safety properties for infinite-state systems which are *monotonic* w.r.t. a *well quasi-ordering* on the set of configurations. The main idea is to perform symbolic backward reachability analysis to check safety properties for such systems.

Outline. We present in Section 2 and Section 3 the type of programs we consider and how we extract a model from them. In Section 4, we state the class of safety properties we want to check, and in Section 5 a solution. We finish with a small description of our experimental results in Section 6 and a conclusion.

2. Language

We analyze concurrent programs written in a stripped version of the C language and using POSIX threads. We call it here the SML language (Simple Multithreaded Language). We use a relatively small set of operations which follows closely those of an instruction set architecture (ISA). Intuitively, we filter out the C language and Pthreads constructs. An ISA includes arithmetic (e.g. *add* and *sub*) and logic instructions (e.g. *and*, *or*, *not*), data instructions (e.g. *load*, *store*) and control flow instructions (e.g. *goto*, *branch*). Arithmetic and logic instructions are simply pure CPU operations. We are interested in instructions touching the main memory and instructions controlling the flow. Hence, the language we allow abstracts away the CPU operations and narrows down the operations to a set of *movers*, i.e. loading from and storing to the main memory and operations related to controlling the flow, thread synchronization and thread bookkeeping.

We do not define formally the semantics of SML programs. They allow multiple threads to execute concurrently and manipulate two kinds of variables: local and global. A global variable is shared by all threads. A local variable is local to a given thread and cannot be accessed by other threads. A shared variable x is a global variable that can be read and written by any thread. We abstract away the data and extract the mover instructions as *Read x* or *Write x* accordingly. Local variables are useful for the control flow only and therefore abstracted away. The *if*, *if-then-else* and *while* statements are allowed in the form of *branch* and *goto* combinations. *for*-loops can be unrolled and equivalently implemented with *while* and *goto*.

Threads can use locks for synchronization purposes through *acquire* and *release* primitives. A lock (also called *mutex*) is a special shared variable that has two values: it is either *free* or *busy*. A thread trying to acquire a lock will block if the lock is busy. It will acquire the lock (i.e. atomically set it to busy) if the lock was free and continue execution. Releasing the lock resets it to free.

Threads can use condition variables for synchronization purposes through *wait* and *signal* primitives. According to the POSIX semantics, wait shall be called while holding a lock or undefined behavior results. If a thread waits on a condition variable, it releases the lock and blocks its execution. A thread will try to re-acquire the lock (leading to eventual delay) and resume execution if the condition variable is signaled by another thread. We do not allow at the moment to broadcast a wake up signal (i.e. wake up all threads waiting on the condition variable). Signaling a condition variable, while no thread is waiting on that condition variable, has no effect. The wait instruction suffers from spurious wakeups as it may return when no thread specifically signalled that condition variable. We do not disallow this behaviour and will handle it in our model. It simply forces the programmer to check a predicate invariant upon return and it is not orthogonal to our analysis.

```
int counter;
pthread_mutex_t L;

pthread_mutex_lock(L);
counter++;
pthread_mutex_unlock(L);
```

```
shared counter, L;

acquire L;
read counter;
write counter;
release L;
```

Figure 1. Critical section problem in pthreaded code (left) and its SML counterpart (right).

```
int buffer; pthread_mutex_t L;
pthread_cond_t cvEmpty, cvFull;

//Many Producers
pthread_mutex_lock(L);
while(true){ /*branch*/
  pthread_cond_wait(cvEmpty,L);
  buffer = data;
  pthread_cond_signal(cvFull);
 }
pthread_mutex_unlock(L);


//Many Consumers
pthread_mutex_lock(L);
while(true){ /*branch*/
  pthread_cond_wait(cvFull,L);
  val = buffer;
  pthread_cond_signal(cvEmpty);
 }
pthread_mutex_unlock(L);
```

```
shared buffer, L, cvEmpty, cvFull;

//Many Producers
acquire L;
while(true){ /*branch*/
  wait cvEmpty, L;
  write buffer;
  signal cvFull;
 }
release L;


//Many Consumers
acquire L;
while(true){ /*branch*/
  wait cvFull, L;
  read buffer;
  signal cvEmpty;
 }
release L;
```

Figure 2. Producers/Consumers in pthreaded code (left) and its SML counterpart (right).

Finally, a thread can create another thread and continue its execution. We do not allow at the moment to wait for termination of newly created threads.

Figure 1 and Figure 2 show examples written in C using Pthreads and its equivalent using the SML language. We do not show the whole code but just the part of interest. Figure 1 shows a critical section problem where multiple threads try to update a counter. Only one thread is allowed to update the counter at a time, to avoid a data race. Figure 2 is a producers/consumers example. The producers update a shared buffer, if the buffer is empty and the consumer reads the buffer if the buffer is full.

3. Model

3.1. Petri Nets

A Petri net $\mathcal{N}$ is a tuple (P, T, F) where P is a finite set of *places*, T is a finite set of *transitions* and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, such that $P \cap T = \emptyset$.

If $(p,t) \in F$, p is said to be an *input* place of t and if $(t,p) \in F$, p is said to be an *output* place of t. We use $I(t) = \{p \in P |\ (p,t) \in F\}$ and $O(t) = \{p \in P |\ (t,p) \in F\}$ to denote the sets of input places and output places of t respectively.

A *configuration* c of a Petri net, often called a *marking* in the literature, is a multiset over P and represents a valuation of the number of *tokens* in each place.

The transition system induced by a Petri net consists of the set configurations together with the transition relation defined on them. The operational semantics of a Petri net is defined through the notion of *firing* transitions. This gives a transition relation on the set of configurations. More precisely, a transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. A transition is *enabled* if each of its input places has at least one token. A transition may fire if it is enabled. [1]Formally, when a transition t is enabled, we write $c \xrightarrow{t} c'$ if c' is the result of firing t on c. We define $\longrightarrow$ as $\bigcup_{t \in T} \xrightarrow{t}$ and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets C_1 and C_2 of configurations, we use $C_1 \longrightarrow C_2$ to denote that $c_1 \longrightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$.

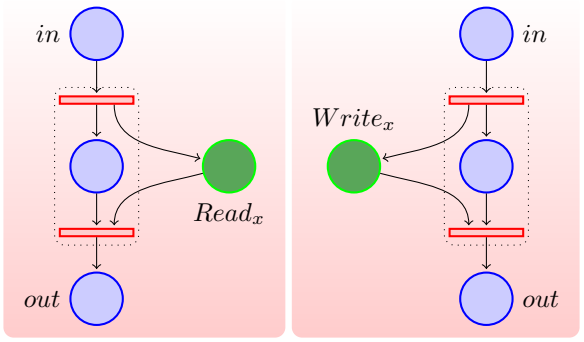
3.2. Modeling programs using Petri Nets

We model the flow of control in SML programs using Petri Nets. A concurrent program contains multiple separate threads of control and each thread is assigned a particular task, so-called *job type*, modeled by a petri net. SML programs have a finite number of job types. Multiple instances of a job type will be modeled by multiple tokens. Note that the structure of the petri net is then static.

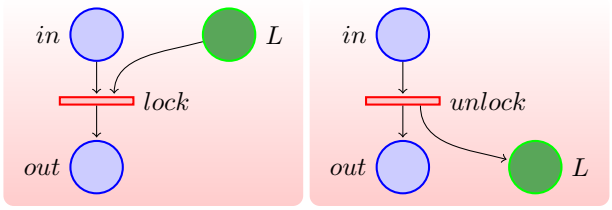
Transitions in the petri net correspond to thread statements and places are used to control the flow and the shared variable dependencies. This modeling formally captures the concurrency between threads using the concurrency constructs of a petri net, captures synchronization between threads (e.g. locks, access to shared variables, condition variables, ...) using appropriate mechanisms in the net, and formalizes the fact that data is abstracted in a sound manner. In the following, a place will be represented by a ○ and by a ● when it is shared. A transition will be represented as ▭.

[1]Petri nets allow multiple arcs from a place to a transition (and vice-versa), where as many tokens are removed (or created) when an enabled transition fires. But we will not use that construct.

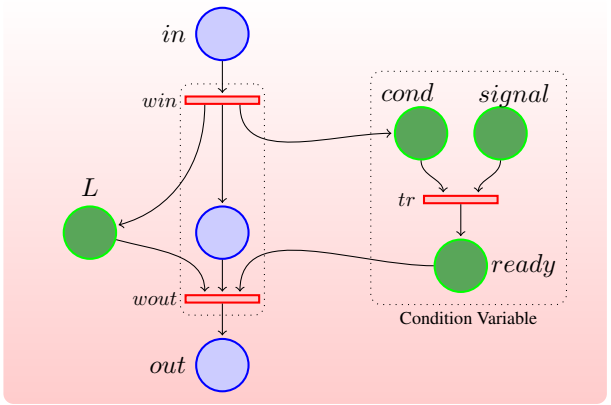
Reading and Writing a shared variable. A shared variable v is associated with two places, $Read_v$ and $Write_v$. A thread places a token in $Read_v$ (resp. $Write_v$) if it is currently accessing the variable v for reading (resp. writing). We model read and write accesses to shared variables with two transitions.



Acquiring and releasing a lock. There is a place L associated with each lock. Intuitively, if L contains a token, the lock is free, otherwise it is busy. This ensures that only one thread can hold the lock at a time. Note that L is a global variable.

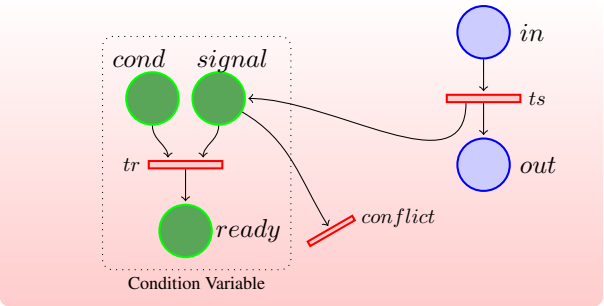


Waiting on a condition variable. A condition variable is modeled with 3 places, namely *cond*, *signal* and *ready*, and 3 transitions as follows.



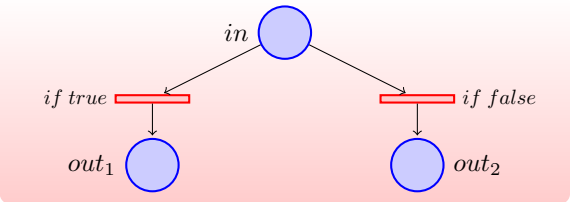
The transition *win* releases the lock and places a token in the *cond* place. The thread is then blocked since neither *wout* nor *tr* are enabled. If another thread places a token in *signal*, *tr* is enabled. If *tr* fires, the blocked thread is ready to fire *wout* to resume execution, with an eventual delay for (re-)acquiring the lock.

Signaling a condition variable. A thread can wake up another blocked thread by placing a token in the *signal* place.



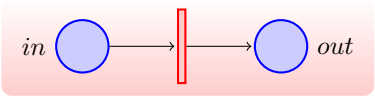
Signaling a condition variable in the model while no thread is waiting on it should have no effect. That is to say, the token in *signal* should be consumed if the *cond* place is empty. To alleviate the problem, we use a *conflict* transition. Recall that an enabled transition only *may* fire. As it is not possible to model the firing of a transition based on the condition that a place is empty, we introduce an abstraction where signals might be lost (i.e. signaling a condition variable might not wake up other threads which are waiting on that condition variable). Nevertheless, it is an over-approximation so we do not bias correctness.

Branching. The *if*, *if-then-else* and *while* statements are easily implemented with *branch* and *goto*. We therefore only model those latter.

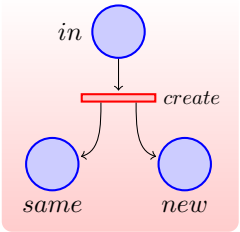


Since we abstract away all data, we cannot determine the branching based on a predicate evaluation. So we use an abstraction where the program takes both branches and model it with two transitions. This eventually introduces false behaviours, and increase the number of false positives. Nevertheless, it is again an over-approximation and does not bias the correctness argument. If, for example, the predicate in the while loop evaluates while reading some shared variable (e.g. $while(x < 10)$), we would model it as a cascade of a read instruction followed by a branch.

Goto or jump. This is only introduced as a convenience to work in conjunction with *branch* and model the control flow.



Creating a new thread.
A new thread is associated with a job type and an initial place. Firing the *create* transition produces a new token that we place in the initial place of the new job type. The calling thread will continue its execution. Note that *same* and *new* place can coincide. It would be easy to extend the transitions to multiple arcs.



3.3. An example

As the petri net of a concurrent program written in the SML language can quickly grow in size, we only show a short example presented in Figure 3, which models the program from Figure 1.

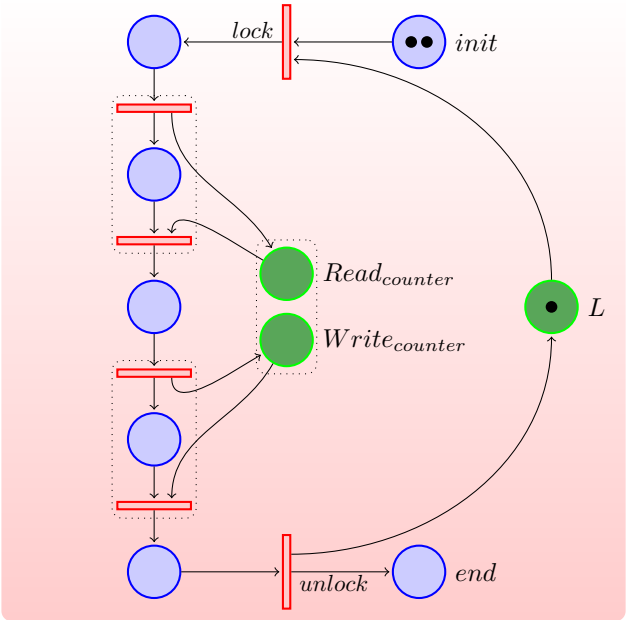
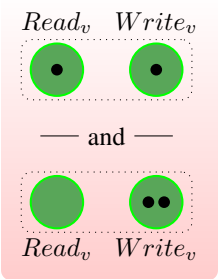


Figure 3. Modeling the critical section problem with petri nets.

4. Race-Freeness

4.1. Bad states and ordering

A data race occurs when multiple threads access a shared variable and at least one has the intention of changing it, without any synchronization constraints (i.e. event ordering). Consequently, given a variable v, we notice that a bad state (i.e. a race on v) is one configuration that contains at least two tokens in either $Read_v$ or $Write_v$ with one of them in the $Write_v$ place.



We therefore introduce the following (partial) pre-order $\preceq$ on configurations. For two configurations c and c', we say that $c \preceq c'$ if all the places in c' contain more tokens than the respective places in c. That is, we can obtain c by removing tokens from c'.

For a configuration c, we use $\widehat{c}$ to denote the *upward closure* of c, i.e. $\widehat{c} = \{c' |\ c \preceq c'\}$. For a set C of configurations, we define $\widehat{C}$ as $\bigcup_{c \in C} \widehat{c}$.

Upward closed sets are attractive to use because they can be characterized by their minimal elements, which often makes it possible to have efficient symbolic representations of infinite sets of configurations.

4.2. Safety property

A set C of configurations is said to be *reachable* if $C_{init} \xrightarrow{*} C$. Checking the safety property amounts to performing reachability analysis: is it possible to reach a bad configuration from an initial configuration through a sequence of actions performed by different threads? It can be shown using standard techniques [18, 11], that checking safety properties can be translated into instances of the following coverability problem.

Coverability
Instance:

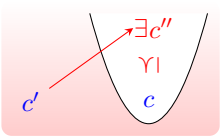
- A set of initial configurations C_{init}.
- An upward closed set of bad configurations C_{fin}.

Question: Is $\widehat{C_{fin}}$ reachable? (i.e. $C_{init} \xrightarrow{*} \widehat{C_{fin}}$?)

The main idea is to perform symbolic backward reachability analysis, using upward closed sets, to check for race-freeness for concurrent programs written in the SML language. A negative answer guarantees the absence of races in the program.

5. Backward Reachability Analysis

Computing predecessors. For a configuration c and a transition t, we define $Pre_t(c) = \left\{c' |\ \exists c'' \in \widehat{c}, s.t.\ c' \xrightarrow{t} c''\right\}$.



$Pre_t(c)$ represents the set of configurations that could reach $\widehat{c}$ by firing t. Notice that the transition system induced by the petri net is monotonic with respect to $\preceq$. Indeed, consider the configurations c_1, c_2 and c_3, such that $c_1 \xrightarrow{t} c_2$ and $c_1 \preceq c_3$, then there exists a configuration c_4 where $c_3 \xrightarrow{t} c_4$ and $c_2 \preceq c_4$. That is to say, if we can fire a transition on a configuration, we can also fire it on a configuration with more tokens in the same places, and the results are also ordered accordingly.

It follows from the anti-symmetry property of $\preceq$ that each upward closed set has a unique generator. Consequently, $Pre_t(c)$ is upward closed and it has a generator. The generator is computed by adding a token to each place in $I(t)$ and by removing a token from each place in $O(t)$ that contained a token (or equivalently removing a token from each output place and resetting to zero the negative values).

We define the backward transition system on configurations, with respect to the pre-order $\preceq$, as follows. For configurations c_1 and c_2, we say that $c_1 \overset{t}{\rightsquigarrow} c_2$ iff $Pre_t(c_1) = \widehat{c_2}$ and we say that $c_1 \rightsquigarrow c_2$ if $c_1 \overset{t}{\rightsquigarrow} c_2$ for some $t \in T$. Note that it makes all transitions always backwards enabled. We define $Pre(c) = \bigcup_{t \in T} Pre_t(c) = \{c' |\ c \rightsquigarrow c'\}$ and $Pre(C)$ as $\bigcup_{c \in C} Pre(c)$. $Pre(C)$ represents the set of all configurations that could reach C by firing a transition in the petri net. Note that, for a finite set C, $Pre(C)$ is an infinite set that can be represented by a finite set of minimal configurations (w.r.t. $\preceq$).

Algorithm. Starting from the finite set C_{fin}, we define the sequence $I_0, I_1, I_2, \ldots$ of sets by $I_0 = \widehat{C_{fin}}$ and $I_{j+1} = I \cup Pre(I_j)$. Intuitively I_j denotes the set of configurations from which C_{fin} is reachable in at most j steps. Thus if we defined $Pre^*(C_{fin})$ to be $\bigcup_{j \geq 0} I_j$, then C_{fin} is reachable if and only if $C_{init} \cap Pre^*(C_{fin}) \neq \emptyset$.

For a set A, we say that the pre-order $\sqsubseteq$ is a *well quasi-ordering (WQO)* on A if the following property is satisfied: for any infinite sequence $a_0, a_1, a_2, \ldots$ of elements in A, there are i, j such that $i < j$ and $a_i \sqsubseteq a_j$. Since we have $I_0 \subseteq I_1 \subseteq I_2 \subseteq \ldots$, and the pre-order $\preceq$ is WQO by Dickson's lemma, it follows by [2] that there is a k such that $I_k = I_{k+1}$ and hence $I_l = I_k$ for all $l \geq k$, implying that $Pre^*(C_{fin}) = I_k$. Each I_j is an infinite set, but we know that we can represent it by a finite set of (minimal) configurations.

The algorithm is guaranteed to terminate.

Input: Two sets C_{init} and C_{fin} of configurations.
Output: $C_{init} \xrightarrow{*} \widehat{C_{fin}}$?

```
WorkList := C_fin
Explored := ∅
while (WorkList ≠ ∅) {
    remove some c from WorkList
    if ∃c' ∈ C_init, c ⪯ c' {
        return true
    } else if ∃c' ∈ Explored, c' ⪯ c {
        discard c
    } else {
        WorkList := WorkList ⋃ Pre(c)
        Explored :=
        {c} ⋃ {c' | c' ∈ Explored ∧ (c ⋠ c')}
    }
}
return false
```

Figure 4. Algorithm outline

6. Experiments

We present a few examples on which we applied the method. As the equivalent petri net for each example can quickly grow, we narrowed down the examples to a characteristic part, namely to test the presence or absence of data races. Some examples are classical examples of synchronization disciplines. We implemented a small prototype in Java and report in table 1 the results of the runs. We display the number of configurations kept at any time in the analysis (#Conf.), the number of configurations that have been subsumed (#Subsum., i.e. discarded by the algorithm because already simulated by other smaller configurations), and the number of iteration of the algorithm (#Iter.). Note that we also tested some examples that did contain a data race. We also added whether the analysis found a race or not. All examples ran in less than a second[2].

The `Counter` and `CounterWithLock` examples represent a shared counter which is incremented as depicted in Figure 1. The `CheckThenAct` and `CheckThenAct-Lock` examples represent a classical race condition described in Section 1 and depicted in Figure 5. Figure 2 depicts the `Prods/Cons` (`Prods/Cons 2` is another variant) as the classical producer/consumer programming style. Note that the `Lock-ReadWriteOnly` example shows a program that secures only the read and write accesses to shared variable. A thread in that program can indeed read a shared variable, store it in a local variable, change the local variable as it wishes, and store the result back into the shared variable. While there is no data race in that program, it is not a good programming practice, as the shared variable might have been updated, and the first read of the shared variable does not reflect its actual value. It has indeed been argued in [10] that the absence of data race is not a strong enough condition.

[2]Performance is not exactly the focus in this paper, nor are limitations, but we wanted to show that it was not a bottleneck.

```
//Thread A                      // Thread B

pthread_mutex_lock(L);          if(data > 0){
data++;                            /*do this*/
pthread_mutex_unlock(L);        } else {
                                   /*do that*/
                                }
```

```
//Thread A                      // Thread B
                                pthread_mutex_lock(L);
pthread_mutex_lock(L);          if(data > 0){
data++;                            /*do this*/
pthread_mutex_unlock(L);        } else {
                                   /*do that*/
                                }
                                pthread_mutex_unlock(L);
```

Figure 5. Check then act in pthreaded code. The read in the if statement was not "secured".

Table 1. Experimental Results

Prog.	#Conf.	#Subsum.	#Iter.	Safe?
`Counter`	10	3	4	-
`CounterWithLock`	14	7	6	✓
`CheckThenAct`	20	12	5	-
`CheckThenAct-Lock`	13	4	4	✓
`Prods/Cons`	310	645	19	✓
`Prods/Cons 2`	290	561	17	✓
`Lock-ReadWriteOnly`	25	13	8	✓

7. Conclusion

We have presented a method to model-check race-freeness of programs written in a subset of the C language, using POSIX threads. We model the behaviour of the system as Petri nets and take advantage of upward closure to efficiently represent infinite sets of configurations. Race-freeness is a safety property and we check it using a symbolic backward reachability analysis. It is based on a simple algorithmic principle and is fully automatic and sound. It is moreover guaranteed to terminate for the class of properties we consider.

We plan to extend the model to obtain a closer relationship to pthreaded programs. For instance, we will include waiting for termination of a specific thread and broadcasting of a wake up signal. We also would like to measure

the usefulness of the method by examining the amount of false positives and the limitations with respect to the program size. Finally, we would like to report more precise program traces involving race conditions.

References

[1] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. LICS '96,* 11^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.

[2] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[3] P. A. Abdulla and A. Nylén. Timed Petri nets and BQOs. In *Proc. ICATPN'2001: 22nd Int. Conf. on application and theory of Petri nets*, volume 2075 of *Lecture Notes in Computer Science*, pages 53 –70, 2001.

[4] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Roby. Bandera: a source-level interface for model checking java programs. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 762–765. ACM, 2000.

[6] G. Delzanno. Automatic verification of cache coherence protocols. In Emerson and Sistla, editors, *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer Verlag, 2000.

[7] J. Dingel. Computer-assisted assume/guarantee reasoning with verisoft. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 138–148. IEEE Computer Society, 2003.

[8] E. Emerson and K. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. LICS '98,* 13^{th} *IEEE Int. Symp. on Logic in Computer Science*, pages 70–80, 1998.

[9] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. LICS '99,* 14^{th} *IEEE Int. Symp. on Logic in Computer Science*, 1999.

[10] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12. ACM, 2003.

[11] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[12] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13. ACM, 2004.

[13] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190. ACM, 2003.

[15] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24. ACM, 2004.

[16] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer Verlag, 1982.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37. ACM, 1997.

[18] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. LICS '86,* 1^{st} *IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[19] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234. ACM, 2005.

Paper session 4: Library suppport for multicore computing

NOBLE: Non-Blocking Programming Support via Lock-Free Shared Abstract Data Types

Håkan Sundell
School of Business and Informatics
University of Borås, 501 90 Borås
E-mail: Hakan.Sundell@hb.se
&
Parallel Scalable Solutions AB
Box 916, 501 10 Borås
E-mail: phs@pss-ab.se

Philippas Tsigas
Department of Computer Science and Engineering
Chalmers University of Technology, 412 96 Göteborg
E-mail: tsigas@chalmers.se

Abstract

An essential part of programming for multi-core and multi-processor includes efficient and reliable means for sharing data. Lock-free data structures are known as very suitable for this purpose, although experienced to be very complex to design. In this paper, we present a software library of non-blocking abstract data types that have been designed to facilitate lock-free programming for non-experts. The system provides: i) efficient implementations of the most commonly used data types in concurrent and sequential software design, ii) a lock-free memory management system, and iii) a run time-system. The library provides clear semantics that are at least as strong as those of corresponding lock-based implementations of the respective data types. Our software library can be used for facilitating lock-free programming; its design enables the programmer to: i) replace lock-based components of sequential or parallel code easily and efficiently, ii) use well-tuned concurrent algorithms inside a software or hardware transactional system. In the paper we describe the design and functionality of the system. We also provide experimental results that show that the library can considerably improve the performance of software systems.

1. Introduction

Explicit multi-threading is an efficient way to exploit the offered parallelism of multi-core and multi-processor based systems. Fundamental to this paradigm is the ability to share data among the threads. To avoid inconsistency of the shared data due to concurrent modifications, accesses to the shared data must be protected and the common solution using mutual exclusion is known for several serious problems. The alternative of using non-blocking synchronization can avoid these problems and lock-free data structures have been shown to permit substantial performance improvement of parallel applications[19] and have also been of interest to designers of languages as C++ [1] and Java.

Two basic non-blocking methods have been proposed in the literature; *lock-free* and *wait-free* [4]. Lock-free implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses, some of them might have to repeat the operation in order to correctly complete it. However, real-time systems might have stronger requirements on progress, and thus in wait-free implementations each task is guaranteed to correctly complete any operation in a bounded number of its own steps, regardless of overlaps of the individual steps and the execution speed of other processes; i.e. while the lock-free approach might allow (under very bad timing) individual processes to starve, wait-freedom strengthens the lock-free condition to ensure individual progress for every task in the system.

Large efforts have recently been on generalized methods for designing non-blocking data sharing as Software Transactional Memory, although still recognized [7] to be far from as efficient as ad-hoc designed algorithms. However, designing ad-hoc lock-free data structures is known to be very complex and can thus only be done safely by experts. For example, although many promising and practical oriented scientific results on non-blocking data structures [20, 10, 3, 8, 9] have appeared in the literature, these have still not migrated much into practice. In an attempt to make non-blocking synchronization a realistic alternative for practitioners, we have created a software library contain-

ing a large number of lock-free implementations of common data structures. This task is also a natural continuation of our previous experience with designing and implementing non-blocking algorithms [18, 15, 2, 16, 17].

NOBLE offers a library support for non-blocking multiprocess synchronization in shared memory systems. NOBLE has been designed in order to: i) provide shared data structures for the use of non-experts, ii) offer an orthogonal support for synchronization in respect to the offered functionality, iii) be easily portable, and iv) contain efficient implementations. We will throughout the paper illustrate the features of NOBLE using the C and C++ languages, although other languages can easily be supported.

The rest of the paper is organized as follows. Section 2 described the motivation for creating the library and in Section 3 the overall design is described. Section 4 describes the programmer's interface to NOBLE. In Section 5, some benchmark experiments are shown. Finally, Section 6 concludes this paper.

2. Motivation and Guidelines

Implementation of the published algorithms for non-blocking data structures are not straight-forward "from the box" as they require thorough knowledge in several aspects:

- **Non-blocking memory management**. Dynamic data structures require memory management. Concurrent data structures consequently require concurrent memory management, which is mostly supported in the system as memory allocation only and implemented using locks, letting to the user to handle when and where to actually allocate or free memory. Language specific solutions include garbage collection facilities. However, non-blocking algorithms need non-blocking memory management in order to be fully lock-free or wait-free, as using locks in any sub operation would invalidate this property. Moreover, the data structure needs to be interoperable with the surrounding code of the main program. The items that are stored within the data structure will be used and concurrently referenced both in and outside of the data structure, and thus needs to be memory managed for safe access.

- **Memory barriers**. The published algorithms normally assume sequential consistency for the operations on memory. However, due to optimized memory bus models and internal processor optimizations in modern multi-core and multi-processor systems, the individual threads are by default not executing the operations on memory in program order and updates on memory might not be globally visible by other threads in the same order. Consequently, if implementing the published algorithms as is, they will very likely result in failures sooner or later [5]. The necessary methods to enforce stronger consistency are hardware-dependent and done by the means of extra machine instructions which needs to be inserted at the right place for every instruction accessing memory which order needs to be controlled. Unfortunately, these instructions are extremely performance degrading and needs to be used with care.

- **Compiler optimization**. The order in which the steps of the non-blocking algorithm are executed by each processor is crucial for its correctness to allow for every step to be interleaved with any other step possibly executed by another processor. Compiler optimization, which is essential for decent performance, can reorder or replace code normally without any concerns about concurrency or non-blocking effects. Thus, the programmer must ensure the correctness of the resulting machine code, with all the means available, e.g. proper insertions of keywords like volatile etc.

- **The main algorithm**. In order to implement the previous issues both properly and efficient, it is needed to understand all steps of the main algorithm and their intended interactions.

Consequently, a trustworthy implementation needs to be performed by experts. Our motivation was therefore to create a software library of abstract data types aimed for concurrent environment that has the best chances to be adopted by practitioners. Consequently we are aiming for the library to be: (i) efficient, (ii) versatile, (iii) portable, (iv) interoperable, and (v) fast applicable.

A study of available algorithms for concurrent data structures with a practical aim, gives quickly the conclusion that there are no single implementation for each type of data structure that fulfills all of the requested library properties. Implementations can vary in properties as:

- **Time complexity**, e.g. depending on the algorithm used, searches can be done in linear [8], probabilistic logarithmic [16], or deterministic logarithmic.

- **Contention**, e.g. the algorithms can create various amount of contention on the memory subsystem.

- **Overall performance**, e.g. how much work is actually performed in the system per time unit.

- **Scalability**, e.g. how the performance will vary with the increasing number of threads.

- **Space**, e.g. memory requirements can vary significantly with the implementation and number of threads.

- **Semantics**, e.g. some implementations might not allow items to be deleted [20] or updated.

- **Locality**, e.g. some implementations require that each thread keeps local information about the data structure.
- **Dependencies and Limitations**, e.g. might only work together with certain memory managers, hardware architectures, or allows up to certain number of threads.

3. Design and Features of NOBLE

The main architecture of NOBLE is layered, as seen in Figure 1a, and is for efficiency reasons implemented in C and Assembler and is merged together with the actual application at compile-time. The C++ interface is provided as a template library for efficient in-line compilation.

The main layer of NOBLE constitutes of a set of components of various implementations of common abstract data types. This layer encapsulates functionality and fully hides complexity of the underlying shared data structure, and works directly on the hardware where possible. The components for abstract data types are facilitated by the memory layer and the run-time layer. The memory manager layer services both the actual abstract data types as well as the data objects stored inside, with efficient concurrent memory allocation and garbage collection facilities.

The run-time system layer, servicing the abstract data types and memory manager layers, abstract the operating system's functionality for system-wide memory and lock-based synchronization management, as well as facilitating for keeping track of the components and their instances in a multi-thread versus a multi-process environment. This layer also abstracts the processor and memory hardware by supporting a well-defined set of atomic primitives for read, writes and updates, implemented using the specific platform's synchronization [12] and memory barriers.

Thus, this framework enables the library to being easily portable. The implementations for the individual components are written independent of platform, and platform specific code is isolated to a small part of the library with a well defined internal interface.

4 Multithread and Multiprocess Components

Each component has a set of different creator functions that specify the particular implementation, that all result in the same type of object from which operations are called:

```
NBLQueueRoot *queue =
NBLQueueCreateLB(); // (C)
NBL::Queue<T> *queue =
NBL::Queue<T>::CreateLB(); // (C++)
```

As all implementations of a component normally support the same interface to the operations, this design makes it very easy to change, or even select in run-time, which implementation to use, e.g. change from lock-based to lock-free.

Some implementations need to create local data for each thread and also specify the calling thread identity for operations. Instead of specifying the total number of threads and creating local data for each at the time when creating the object, dynamic attachment of threads to the component object are supported by the means of a handle. The handle needs to be created by each thread for the specific component object, see Figure 1b, and is the only way to call the operations:

```
NBLQueue *handle =
NBLQueueCreateHandle( queue ); // (C)
void *item = NBLQueueDequeue( handle );
// (C)
```

The C++ interface, causing a slight overhead, completely abstracts the handle creation and allows the direct calling of operations from the component object:

```
T* item = queue->Dequeue(); // (C++)
```

4.1. Semantics

The semantics of the components, which has been designed to be the very same for all implementations of a particular abstract data type, are based on the sequential semantics of common abstract data types and adopted for concurrent use. The set of operations has been limited to those which can be practically implemented using both non-blocking and lock-based techniques. Due to the concurrent nature, also new operations have been added, e.g. `Update` which cannot be replaced by `Delete` followed by `Insert`. Some operations also have stronger semantics than the corresponding sequential, e.g. traversal in the List is not invalidated due to concurrent deletes, compared to the iterator invalidation in STL. As the published algorithms for concurrent data structures often diverges from the chosen semantics, a large part of the implementation work in NOBLE, besides from adoption to the framework, also constitutes of considerable changes and extensions to meet the awaited semantics.

4.2. Components Overview

The abstract data types, see Table 1, each have several implementations that are named according to their characteristics. Naturally, LF stands for lock-free, WF stands for

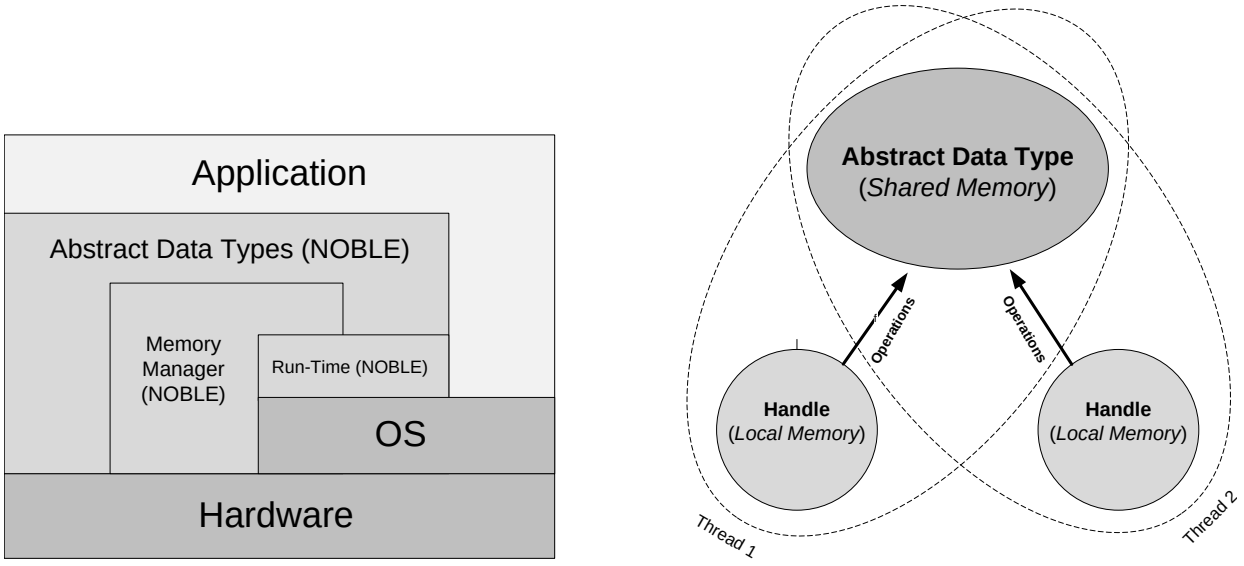


Figure 1. (a) The architecture of the NOBLE library system. (b) Operations per thread basis are performed through local handles that access and modify the shared data structure.

wait-free, and LB stands for lock-based. Concerning the memory requirements of the abstract data types, B stands for bounded, and U stands for unbounded memory usage. The Queue can be based on either a static (S) or dynamic (D) underlying data structure. The Deque can either offer high (H) or limited (L) level of parallelism. The Priority Queue and Dictionary can offer an expected logarithmic (E), deterministic logarithmic (D), or linear (L) time complexity for searches with respect to their size. The List can be either singly (S) or doubly (D) linked.

The implementations of the other data structures, see Table 2, are named similar to the abstract data types. The Snapshot can support a single (S) scanner together with either single (S) or multiple (M) updaters. The implementations of the atomic word object support a subset of the functionality, either basic (B) or extended functionality of either multi-word updates (CASN) or load-linked updates (LL). The memory manager implementations either support fixed-size, a selection of sizes (C), or arbitrary size (H) of memory blocks for allocation. The memory reclamation mechanism can either be strong (S), medium (M), or weak (W) dependent on the ability to keep watch over both global and local pointers. The number of local pointers that can be handled by each thread can either be limited (L) or unlimited (U).

4.3. Interoperability

As the library stores the items in the abstract data types by reference, the user program sometimes also needs to consider when to reclaim the item's memory. For abstract data types as the Stack, Queue, and Deque this is no problem as only two threads (the one that puts it and the one that removes it) access each item. However, for the Dictionary some threads might access the items through searching at the same time as another thread might remove it from the dictionary. To easily facilitate safe handling of the user items, the Memory component can be used for allocation and referencing to the items:

```
void *item = NBLMemoryAllocBlock(
memoryHandle );
```

The abstract data type component also needs to be aware of which memory object to use for the items. As all component support functions for setting run-time parameters, the memory object can be specified using one of the parameters:

```
NBLDictionarySetParameter( dictionary,
NBL_PARAM_VALUE, memory );
```

By setting the memory parameter, the dictionary can return references to the items that are safe to access even if the items have been concurrently removed instantly afterwards.

Table 1. Components overview, part 1(2)

Component	Implementations	Operations
Stack	LF_B, LF_U, LB	*ok*=Push(*item*) *item*=Pop()
Queue	WF_SS, LF_DB, LF_DU, LF_SB, LB	*ok*=Enqueue(*item*) *item*=Dequeue()
Deque	LF_HB LF_HU LF_LB LB	*ok*=PushLeft(*item*) *ok*=PushRight(*item*) *item*=PopLeft() *item*=PopRight()
PQueue	LF_EB, LF_EU, LB_SD, LB_DD, LB_E	*ok*=Insert(*priority*,*item*) *item*=DeleteMin(ref *priority*) *item*=FindMin(ref *priority*)
Dictionary	LF_EB LF_EU LF_LB LB_E	*ok*=Insert(*key*,*item*) *ok*=Update(*key*,*item*,ref *olditem*) *item*=Delete(*key*) *item*=Find(*key*)
List	LF_SU LF_DB LF_DU LB_S LB_D	*ok*=InsertBefore(*item*) *ok*=InsertAfter(*item*) *item*=Delete() *item*=Read() First() Last() *ok*=Next() *ok*=Previous()

After accessing the item, the reference has to be released:

```
void *item = NBLDictionaryFind(
dictHandle, key );
NBLMemoryReleaseRef( memoryHandle, item
);
```

The Priority Queue and Dictionary support besides integers for priorities and keys also user-defined data types for these. As the abstract data types perform searches among the stored items, these user-defined data types need to be enumerable and thus comparable. The user-defined function for comparison of the user-defined data types are specified using the run-time parameters to the component objects. In the C++ interface this is automatically handled by the user defining the $<$ and $==$ operators for the specific data type.

Some implementations uses back-off techniques to increase performance in contention intensive situations. For best performance these timings are platform dependent, and can thus be tweaked by the means of the on-line parameters.

If needed, the user can also tune the whole library's use of the system's default memory manager for pre-allocation and its functionality for handling mutexes and semaphores, by specifying user-defined functions for replacement.

NOBLE provides efficient implementations of the most commonly used data types in concurrent and sequential software design with an object-oriented API. A list of the data types currently provided by NOBLE as well as the NOBLE API can be found in [13].

4.4. Related Work

Commercially and by communities, there have been several attempts of incorporating non-blocking data structures and synchronization into frameworks and libraries, although mostly only to a comparably small extent. To the best of our knowledge, the major libraries with significant non-blocking content are the following:

- Intel Threading Building Blocks [6] This is a framework for implicit/explicit parallel programming in C++, also containing a collection of container classes. The containers are designed using a combination of lock-free and lock-based techniques. The containers are Queue, Vector or HashMap. Items in the containers are stored by value.
- Java Concurrency Package [14] This is a library package for explicit multi-thread programming in Java, containing several thread-safe container classed.

Table 2. Components overview, part 2(2)

Component	Implementations	Operations
Snapshot	WF_SS, WF_SM, WFR_SM, LB	Scan(*values[]*) Update(*component*,*value*)
Word	WF_B WF_CASN LF_LL	*ok*=Init(*address*,*value*) Deinit(*address*) *value*=Read(*address*) Write(*address*,*value*) *value*=Add(*address*,*value*) *value*=Swap(*address*,*value*) *value*=Op(*address*,function,*value*) *ok*=CAS(*address*,*oldvalue*,*newvalue*) *ok*=CASN(*address[]*,*oldval[]*,*newval[]*) *value*=LL(*index*,*address*) *ok*=VL(*index*,*address*) *ok*=SC(*index*,*address*,*value*)
Memory	LF_SLB, LF_SUU, LF_MLB, LF_WLB, LF_CSLB, LF_CSUU, LF_CMLB, LF_CWLB, LF_HSLB, LF_HSUU, LF_HMLB, LF_HWLB, WF_SUU	*address*=AllocBlock() *address*=AllocClass(*sizeClass*) *address*=AllocSize(*size*) DeleteBlock(*address*) *address*=DeRefLink(ref *address*) *address*=CopyRef(*address*) ReleaseRef(*address*) StoreRef(ref *address*,*address*) *ok*=CASRef(ref *address*,*oldaddr*,*newaddr*)

A subset of these is based on lock-free algorithms, e.g., ConcurrentLinkedQueue and ConcurrentHashMap. Items in the containers are by default stored by reference.

- Microsoft Parallel Extensions to the .NET Framework [11] This is a framework for implicit/explicit parallel programming within the .NET framework, also containing a collection of container classes in the Task Parallel Library. The containers based on lock-free algorithms constitute of ConcurrentQueue and ConcurrentStack. Items are stored by value or by reference depending on the type.

NOBLE is a library for explicit multi-thread programming in C or C++. The library contains classes for abstract data types, single and multi-word transactions, and memory management. NOBLE contains 7 types of abstract data types, and offers several lock-free or wait-free implementations of each data type where each implementation suffice different needs and characteristics. Items in the containers are stored by reference.

See Table 3 for a quick overview of the non-blocking abstract data types offered by the mentioned libraries.

5 Experiments

We have performed experiments in order to estimate the performance and illustrate the benefits of having several implementations available for each task. For this purpose we are using micro-benchmarks that test the implementations in different dimensions that affect performance. In this paper, we present the benchmark of having maximum contention, i.e., the concurrent threads are continuously invoking operations.

In our experiments each concurrent thread performs 500 000 randomly chosen sequential operations with equal distribution. Each experiment is repeated several times, and an average execution time for each experiment is estimated. Exactly the same sequence of operations is used for all different implementations compared. All lock-based implementations are based on simple spin-locks. For the queue experiments the initial number of items is zero, and for the dictionary experiments the data structure is initiated with 1000 items. The experiments were performed using different number of threads, varying from 1 to 16.

The results from these experiments are shown in Figure 2, where the average number of operations performed per second (in the whole system, all threads altogether) is drawn as a function of the number of processes. The results clearly indicate that the lock-free implementations in

Table 3. Overview of libraries and contained non-blocking abstract data types.

Library	Languages	Abstract data types
Intel Threading Building Blocks	C++	Queue Vector Dictionary(HashMap)
Java Concurrency Package	Java	Queue Dictionary (HashMap)
Microsoft Parallel Extensions	C# (.NET)	Stack Queue
NOBLE Professional Edition	C C++	Stack Queue Deque Priority Queue Dictionary List Snapshot

NOBLE, largely because of their non-blocking characteristics and partly because of their efficient implementation and memory management, can outperform the respective lock-based implementations significantly.

6. Conclusions

NOBLE is a library for multi-core and multiprocessor data sharing, including implementations of several fundamental and commonly used abstract data types. The library is easy to use and is well suited for explicit multi-thread programming. Thanks to several available implementations and synchronization mechanisms, programs can be easily tuned, even in run-time, to meet specific performance and characteristics demands. Experiments show that the non-blocking implementations in NOBLE offer significant improvements in performance, especially on multi-processor platforms. The library currently supports five architectures (Intel x86, AMD x64, PowerPC, Sparc and Mips), and five operating systems (Windows, Linux, Solaris, Irix and AIX).

The second version of NOBLE is proprietary and made available through a subsidiary company related with our research. Earlier versions available freely for research and teaching is available through http://www.noble-library.org. Future work of NOBLE is to include even more abstract data types and functionality, develop native interfaces for managed languages like C#, and adapt to emerging multi-core and multiprocessor architectures.

References

[1] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS '06)*, Lecture Notes in Computer Science, pages 142–156. Springer Verlag, 2006.

[2] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 202–207. IEEE, Dec. 2005.

[3] T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium of Distributed Computing*, pages 300–314, Oct. 2001.

[4] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[5] L. Higham and J. Kawash. Impact of instruction re-ordering on the correctness of shared-memory programs. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 25–32. IEEE, Dec. 2005.

[6] Intel. *Intel Threading Building Blocks 2.1*, Sept. 2008.

[7] P. E. McKenney, M. M. Michael, and J. Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. In *Proceedings of the 4th workshop on Programming languages and operating systems (PLOS '07)*, pages 1–5. ACM, 2007.

[8] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

[9] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), Aug. 2004.

[10] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.

[11] Microsoft. *Microsoft Parallel Extensions to .NET Framework 3.5, June 2008 Community Technology Preview*, Sept. 2008.

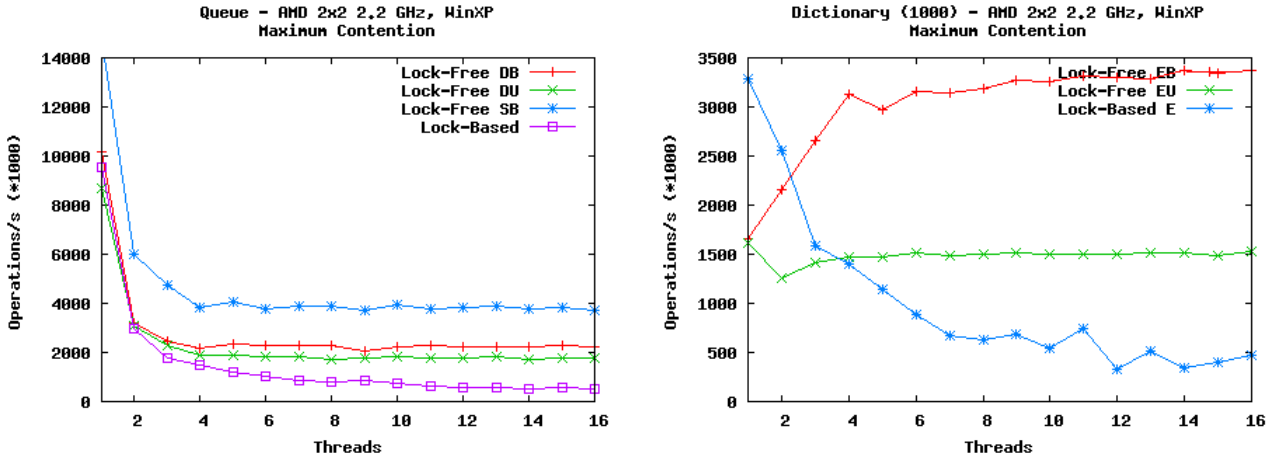


Figure 2. Experiments on a 4-way AMD processor system.

[12] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 1997.

[13] Parallel Scalable Solutions AB. *NOBLE Professional Edition: Application Programmers Interface*, Sept. 2008.

[14] Sun. *Package java.util.concurrent (Java Platform SE 6)*, Sept. 2008.

[15] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 19th ACM Symposium on Applied Computing*, pages 1438–1445. ACM press, Mar. 2004.

[16] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, May 2005.

[17] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

[18] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*, pages 134–143. ACM press, 2001.

[19] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance*, pages 55–67. ACM Press, 2002.

[20] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.

LFTHREADS: A lock-free thread library[a]

Anders Gidenstam
Algorithms and Complexity
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany.
andersg@mpi-inf.mpg.de

Marina Papatriantafilou
Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden.
ptrianta@cs.chalmers.se

Abstract

This paper presents the LFTHREADS*, a thread library entirely based on lock-free methods, i.e. no spin-locks or similar synchronization mechanisms are employed in the implementation of the multithreading. Since lock-freedom is highly desirable in multiprocessors/multicores due to its advantages in parallelism, fault-tolerance, convoy-avoidance and more, there is an increased demand in lock-free methods in parallel applications, hence also in multiprocessor/multicore system services.* LFTHREADS *is the first thread library that provides a lock-free implementation of blocking synchronization primitives for application threads; although the letter may sound like a contradicting goal, such objects have several benefits: e.g. library operations that block and unblock threads on the same synchronization object can make progress in parallel while maintaining the desired thread-level semantics and without having to wait for any "slow" operations among them. Besides, as no spin-locks or similar synchronization mechanisms are employed, memory contention can be reduced and processors/core are able to do useful work. As a consequence, applications, too, can enjoy enhanced parallelism and fault-tolerance. For the synchronization in* LFTHREADS *we have introduced a new method, which we call* responsibility hand-off *(RHO), that does not need any special kernel support. The RHO method is also of independent interest, as it can also serve as a tool for lock-free token passing, management of contention and interaction between scheduling and synchronization.*

[a] An earlier version of this work appeared in OPODIS'07 [8]; a more extended version as technical report in [9].

1 Introduction

Multiprogramming and threading allow the processor(s) to be shared efficiently by several sequential threads of control. Here we study synchronization issues and algorithms for realizing standard thread-library operations and objects (create, exit, yield and mutexes) based entirely on *lock-free* methods.

The rationale in LFTHREADS is that active processors or cores should always be able to do useful work when there are runnable threads available , regardless of what other processors/cores do; i.e. despite others simultaneously accessing shared objects related with the implementation of the LFTHREADS-library and/or suffering stop failures or delays (e.g. from I/O or page-fault interrupts).

Even a lock-free thread library needs to provide blocking synchronization objects, e.g. for mutual exclusion in legacy applications and for other applications where threads might need to be blocked, e.g. to interact with some external device. Our new synchronization method in LFTHREADS implements a mutual exclusion object with the standard blocking semantics for application threads but *without enforcing mutual exclusion among the processors* executing the threads.We consider this an important part of the contribution. It enables library operations blocking and unblocking threads on the same synchronization object to make progress in parallel, while maintaining the desired thread-level semantics, without having to wait for any "slow" operation among them to complete. We achieved this via a new synchronization method, which we call *responsibility hand-off* (RHO), which may also be useful in lock-free synchronization constructions in general, e.g. for *token-passing, contention management and interplay between scheduling and synchronization*. The method is lock-free and manages thread execution contexts without needing special kernel or scheduler support.

Related and motivating work

A special kernel-level mechanism, called *scheduler activations*, has been proposed and studied [2, 6], to enable user-level threads to offer the functionality of kernel-level threads with respect to blocking and also leave no processor idle in the presence of ready threads, which is also the goal of LFTHREADS. It was observed that application-controlled blocking and interprocess communication can be resolved at user-level without modifications to the kernel while achieving the same goals as above, but multiprogramming demands and general blocking, such as for page-faults, seem to need scheduler activations. The RHO method and LFTHREADS complement these results, as they provide thread synchronization operation implementations that do not block each other unless the application blocks within the same level (i.e. user- or kernel-level). LFTHREADS can be combined with scheduler activations for a hybrid thread implementation with minimal blocking.

To make the implementation of blocking mutual exclusion more efficient, operating systems that implement threads at the kernel level may split the implementation of the mutual exclusion primitives between the kernel and user-level. This is done in e.g. Linux [7] and Sun Solaris [30]. This division allows the cases where threads do not need to be blocked or unblocked, to be handled at the user-level without invoking a system call and often in a non-blocking way by using hardware synchronization primitives. However, when the calling thread should block or when it needs to unblock some other thread, an expensive system call must be performed. Such system calls contain, in all cases we are aware of, critical sections protected by spin locks.

Although our present implementation of LFTHREADS is entirely at the user-level, its algorithms are also well suited for use in a kernel - user-level divided setting. With our method a significant benefit would be that there is no need for spin locks and/or disabling interrupts in either the user-level or the kernel-level part.

Further research motivated by the goal to keep processors busy doing useful work and to deal with preemptions in this context includes: mechanisms to provide some form of control on the kernel/scheduler to avoid unwanted preemption (cf. e.g. [20, 18]) or the use of some application-related information (e.g. from real-time systems) to recover from it [5]; [4] and subsequent results inspired by it focusing on scheduling with work-stealing, as a method to keep processors busy by providing fast and concurrent access to the set of ready threads; [28] aims in a similar direction, proposing thread scheduling that does not require locking (essentially using lock-free queuing) in a multithreading library called Lesser Bear; [37] studied methods of scheduling to reduce the amount of spinning in multithreaded mutual exclusion; [38] focuses on demands in real-time and embedded systems and studies methods for efficient, low-overhead semaphores; [1] gives an insightful overview of recent methods for mutual exclusion.

Figure 1 The Compare-And-Swap (CAS) and Fetch-And-Add (FAA) atomic primitives.

```
function
    CAS(address : pointer to word;
        oldvalue : word;
        newvalue : word) : boolean
atomic do
    if *address = oldvalue then
        *address := newvalue;
        return true;
    else return false;
```

```
function
    FAA(addr: pointer to integer;
        increment: integer): integ
atomic do
    ret := *addr;
    *addr := ret + increment;
    return ret;
```

There has been other work at the operating system kernel level [24, 23, 12, 13], where basic kernel data structures have been replaced with lock-free ones with both performance and quality benefits. There are also extensive interest and results on lock-free methods for memory management (garbage collection and memory allocation, e.g. [36, 26, 25, 10, 11, 15]).

The goal of LFTHREADS is to implement a common thread library interface, including operations with blocking semantics, in a lock-free manner. It is possible to combine LFTHREADS with lock-free and other non-blocking implementations of shared objects, such as the NOBLE library [32] or software transactional memory constructions (cf. e.g. [22, 29]).

The paper is organized as follows: first we present the system model together with some background information on lock-free synchronization and the problem we focus on including the application programming interface of LFTHREADS (Section 2); followed by a detailed description of the algorithmic design (Section 3); the correctness of the above (Section 4); some implementation-related information and an experimental study (Section 5). We conclude in Section 6.

2 Preliminaries

System model

The system consists of a set of processors or cores, each of which may have its own local memory as well as it is connected to a shared memory through an interconnect network. Each processor executes instructions sequentially at an arbitrary rate. The shared memory might not be uniform, that is, for each processor the latency to access some part of the memory is not necessarily the same as the latency for any other processor to access that part of the shared memory. The shared memory supports atomic read and write operations of any single memory word, and also stronger single-word synchronization primitives, such

as Compare-And-Swap (CAS) and Fetch-And-Add (FAA) (see Figure 1). These primitives are either available or can easily be derived from other available primitives [19, 27] on contemporary microprocessor architectures. The processors in the system cooperate to run a set of application threads. Each thread consists of a sequence of operations; communication is accomplished via shared-memory operations.

Lock-free synchronization

Lock-freedom [14] is a type of non-blocking synchronization that guarantees that in a set of concurrent operations at least one of them makes progress and thus eventually completes each time. Another type of non-blocking synchronization is *wait-freedom* [21], which guarantees that *every* operation finishes in a finite number of its own steps regardless of the actions of concurrent operations. In the literature we also see *obstruction-freedom* [16], a weak non-blocking synchronization option, guaranteeing only that, at any point, a thread that executes *alone* for a sufficiently large but bounded number of steps can complete its operation. Obstruction free algorithms are distinguished from lock-free and wait-free ones: in the former, progress is not guaranteed in presence of concurrency and operations may even abort.

The correctness condition for atomic non-blocking operations is *linearizability* [17]. An execution is *linearizable* if it guarantees that even when operations overlap in time, each of them appears to take effect at an atomic time instant that lies within its respective time duration, such that the effect of each operation is consistent with the effect of its corresponding operation in a sequential execution in which the operations appear in the same order.

Non-blocking synchronization is attractive as it offers a number of advantages over lock-based synchronization: (i) it does not give rise to priority inversion; (ii) it avoids lock convoys; (iii) it provides better fault tolerance (processor stop failures will not corrupt shared data objects); and (iv) it eliminates deadlock scenarios involving two or more threads both waiting for each other. Due to these facts there is extended research literature on lock-free synchronization (c.f. [31] for an overview) as well as on *universal methods* to transform lock-based constructions into lock-free/wait-free ones (e.g. [3, 14, 35]). Besides ensuring the above qualitative properties, it has also been shown, using well-known parallel applications, that *lock-free* methods imply at least as good performance as lock-based ones in several applications, and often significantly better [31, 33]. Wait-free algorithms, as they provide stronger progress guarantees, are inherently more complex and more expensive than lock-free ones. Obstruction freedom implies very weak progress guarantees and can be used e.g. for reference purposes for studying parallelization.

In LFTHREADS the focus is on *lock-free synchronization* due to its combined benefits in progress, fault-tolerance and efficiency potential.

The problem and LFTHREADS's API

The LFTHREADS library defines the following procedures for thread handling[a]:

procedure create(thread : **out** thread_t; main : **in pointer to procedure**);
procedure exit();
procedure yield();

Procedure *create* creates a new thread which will start in the procedure main. Procedure *exit* terminates the calling thread and if this was the last thread of the application/process the latter is terminated as well. Procedure *yield* causes the calling thread to be put on the ready queue and the (virtual) processor that was running it to pick a new thread to run from the ready queue.

For applications that need lock-based synchronization between threads the thread library provides a mutex object. The mutex object supports the operations:

procedure lock(mutex : **in out** mutex_t)
procedure unlock(mutex : **in out** mutex_t)
function trylock(mutex : **in out** mutex_t): **boolean**

Procedure *lock* attempts to lock the mutex. If the mutex is locked already the calling thread is blocked and enqueued on the waiting queue of the mutex. Procedure *unlock* unlocks a mutex if there are no threads waiting in the mutex's waiting queue, otherwise the first of the waiting threads are removed from the waiting queue and made runnable. That thread becomes the new owner of the mutex. Only the thread owning the mutex may call *unlock*. Function *trylock* tries to lock the mutex. If it succeeds (i.e. the mutex was unlocked) true is returned, otherwise false.

3 Detailed description of the LFTHREADS library

3.1 Data structures used in LFTHREADS

In Figure 2 the data structures used in the implementation of the LFTHREADS library are presented. We assume that we have a data type, context_t, where the CPU context of an execution (i.e. thread) can be stored and some operations to manipulate such contexts. These operations, which can be supported by most common operating systems[b], are:

[a] The interface we present here was chosen for brevity and simplicity. Our actual implementation aims to provide a POSIX threads compliant (IEEE POSIX 1003.1c) interface.

[b] For example in systems conforming to the Single Unix Specification v2 (*SUSv2*), such as GNU/Linux, they can be implemented from

(i) *save*(context) stores the state of the current CPU context in the supplied variable and switches processor to a special system context. There is one such context available for each processor. The return value from *save* is true when the context is stored and false when the context is restored.
(ii) *restore*(context) loads the supplied stored CPU context onto the processor. The restored context resumes execution in the (old) call to *save*, returning false. The CPU context that made the call to *restore* is lost (unless it was saved before the call to *restore*).
(iii) *make_context*(context,main) creates a new CPU context in the supplied variable. The new context will start in a call to the procedure main when it is loaded onto a processor with *restore*.

Each thread in the system will be represented by an instance of the thread control block data type, thread_t, which contains a context_t variable that stores the thread's state when it is not being executed on one of the processors.

Further, we also assume that we have a lock-free queue data structure (like e.g. [34]) for pointers to thread control blocks; the queue supports three lock-free and linearizable operations: *enqueue*, *dequeue* and *is_empty* (each with its intuitive semantics). The lock-free queue data structure is used as a building block in the implementation of LFTHREADS. However, as we will see in detail below, additional synchronization methods are needed to make operations involving more than one queue instance lock-free and linearizable.

3.2 Thread operations in LFTHREADS

The general thread operations and variables used in LFTHREADS are shown in Figure 3. The persistent global and per-processor variables consist of the global shared Ready_Queue[c], which contains all runnable threads not currently being executed by any processor, and the per-processor persistent variable Current, which contains a pointer to the thread control block of the thread currently being executed on that processor.

The thread handling operations, whose required functionality was introduced in section 2, work as follows in LFTHREADS:
(i) Operation *create* creates a new thread control block, initializes the context stored in the block and enqueues the new thread on the ready queue.
(ii) Operation *exit* terminates the thread currently being executed by this processor, which then picks another thread to run from the ready queue.

`getcontext(2)`, `setcontext(2)` and `makecontext(3)`, while in other Unix systems `setjump(3)` and `longjmp(3)` or similar could be used.

[c]The Ready_Queue here is a lock-free queue, but e.g. work-stealing [4] could be used.

Figure 2 Thread context and thread queue operations used in LFTHREADS.

```
type context_t is record ⟨implementation defined⟩;

function save(context : out context_t): boolean;
/* Saves the current CPU context and switches to a
 * system context. The call to save returns true when
 * the context is saved and false when it is restored. */
procedure restore(context : in context_t);
/* Replaces the current CPU context with a
 * previously stored CPU context.
 * The current context is destroyed. */
procedure make_context(context : out context_t;
          main : in pointer to procedure);
/* Creates a new CPU context which will wakeup
 * in a call to the procedure main when restored. */

type thread_t is record
      uc : context_t;
/* Thread control block. */

type lf_queue_t is record ⟨implementation defined⟩;

procedure enqueue(queue : in out lf_queue_t;
          thread : in pointer to thread_t);
/* Appends the thread control block thread to
 * the end of the queue. */
function dequeue(queue : in out lf_queue_t;
          thread : out pointer to thread_t): boolean;
/* If the queue is not empty the first thread_t pointer
 * in the queue is dequeued and true is returned.
 * Returns false if the queue is empty. */
function is_empty(queue : in out lf_queue_t): boolean;
/* Returns true if the queue is empty, and
 * false otherwise. */

function get_cpu_id(): cpu_id_t
/* Returns the ID of the current CPU (an integer). */
```

Figure 3 The basic thread operations and shared data in LFTHREADS.

```
/* Global shared variables. */
Ready_Queue : lf_queue_t;

/* Private per-processor persistent variables. */
Current_p : pointer to thread_t;

/* Local temporary variables. */
next : pointer to thread_t;
old_count : integer;
old : cpu_id_t;

procedure create(thread : out thread_t;
            main : in pointer to procedure)
C1    make_context(thread.uc, main);
C2    enqueue(Ready_Queue, thread);

procedure yield()
Y1    if not is_empty(Ready_Queue) then
Y2        if save(Current_p.uc) then
Y3            enqueue(Ready_Queue, Current_p);
Y4            cpu_schedule();

procedure exit()
E1    cpu_schedule();

procedure cpu_schedule()
CI1   loop
CI2       if dequeue(Ready_Queue, Current_p) then
CI3           restore(Current_p.uc);
```

(iii) Operation *yield* saves the context of the thread currently being executed by this processor, enqueues this thread on the ready queue and then picks another thread to run from the ready queue.

In addition to the public *create*, *exit*, *yield* operations, there is an internal operation in LFTHREADS, namely *cpu_schedule*, which is used for selecting the next thread to load onto the processor. If there are no threads waiting for execution in the Ready_Queue, the processor is idle and waits for a runnable thread to appear.

3.3 Blocking thread synchronization in LFTHREADS and the RHO method

To facilitate blocking synchronization among application threads, LFTHREADS provides a mutex primitive, mutex_t. While the operations on a mutex, *lock*, *trylock* and *unlock* have their usual semantics for application threads, they are lock-free with respect to the processors in the system. This implies improved fault-tolerance properties against stop and timing faults in the system compared to traditional spin-lock-based implementations, since even if a processor is stopped or delayed in the middle of a mutex operation all other processors are still able to continue performing operations, *even on the same mutex*. However, note that an individual application thread trying to lock a mutex will be blocked if the mutex has been locked by another application thread. A faulty application can also dead-lock its threads. It is the responsibility of the application developer to prevent such situations.[d]

Mutex operations in LFTHREADS

The mutex_t structure, shown in Figure 4, consists of three fields:

(i) an integer counter, which counts the number of threads that are in or want to enter the critical section protected by the mutex;

(ii) a lock-free queue, where the thread control blocks of blocked threads wanting to lock the mutex when it is already locked can be stored; and

(iii) a hand-off flag, whose role and use will be described in detail below.

The operations on the mutex_t structure are shown in Figure 4. In rough terms, the *lock* operation locks the mutex and makes the calling thread its owner. If the mutex is already locked the calling thread is blocked and the processor switches to another thread. The blocked thread's con-

[d] I.e. here lock-free synchronization guarantees deadlock-avoidance among the operations that are implemented in lock-free manner, but an *application* that uses objects that have blocking semantics (e.g. mutex) of course needs to take care to avoid deadlocks due to *inappropriate use* of blocking operations by its threads.

Figure 4 The lock-free mutex protocol in LFTHREADS.

```
type mutex_t is record
        waiting : lf_queue_t;
        count : integer := 0;
        hand-off : cpu_id_t := null;

procedure lock(mutex : in out mutex_t)
L1      old_count := FAA(&mutex.count, 1);
L2      if old_count ≠ 0 then
            /* The mutex was locked.
             * Help or run another thread. */
L3          if save(Current_p.uc) then
L4              enqueue(mutex.waiting, Current_p);
L5              Current_p := null; /* The thread is now blocked. */
L6              old := mutex.hand-off;
L7              if old ≠ null and not is_empty(mutex.waiting) then
L8                  if CAS(&mutex.hand-off, old, null) then
L9                      dequeue(mutex.waiting, Current_p);
L10                     restore(Current_p); /* Done. */
L11             cpu_schedule(); /* Done. */

function trylock(mutex : in out mutex_t): boolean
TL1     if CAS(&mutex.count, 0, 1) then return true;
TL2     else if GrabToken(&mutex.hand-off) then
TL3         FAA(&mutex.count, 1);
TL4         return true;
TL5     return false;

procedure unlock(mutex : in out mutex_t)
U1      old_count := FAA(&mutex.count, −1);
U2      if old_count ≠ 1 then
            /* There is at least one waiting thread. */
U3          do_hand-off(mutex);

procedure do_hand-off(mutex : in out mutex_t)
H1      loop /* We own the mutex. */
H2          if dequeue(mutex.waiting, next) then
H3              enqueue(Ready_Queue, next);
H4              return; /* Done. */
            else /* The waiting thread is not ready yet! */
H5              mutex.hand-off := get_cpu_id();
H6              if is_empty(mutex.waiting) then
                    /* Some concurrent operation will see/or
                     * has seen the hand-off. */
H7                  return; /* Done. */
H8              if not CAS(&mutex.hand-off, get_cpu_id(), null) then
                    /* Some concurrent operation acquired the mutex. */
H9                  return; /* Done. */

function GrabToken(loc : pointer to cpu_id_t) : boolean
GT1     old := *loc;
GT2     if old = null then return false;
GT3     return CAS(loc, old, null);
```


text will be activated and executed later when the mutex is released by its previous owner.

In the ordinary case a blocked thread is activated by the thread releasing the mutex by invoking *unlock*, but due to fine-grained synchronization, it may also happen in other ways. In particular, note that checking whether the mutex is locked and entering the mutex waiting queue are distinct atomic operations. Therefore, the interleaving of thread-steps can cause situations such that e.g. a thread A finds the mutex locked, but by the time it has entered the mutex queue the mutex has been released, hence A should not remain blocked in the waiting queue. The "traditional" way to avoid this problem is to ensure that at most one processor at a time modifies the mutex state, i.e. by enforcing mutual exclusion among the processors in the implementation of the mutex operations, e.g. by using a spin-lock. In the lock-free solution proposed here, the synchronization required for such cases is managed with a new method, which we call the *responsibility hand-off* (RHO) method. In particular, the thread/processor that is releasing the mutex is able, using appropriate fine-grained synchronization steps, to detect whether such a situation may have occurred and, in response, "hand-off" the ownership (or responsibility) for the mutex to some other thread/processor.

By performing a *responsibility hand-off*, the processor executing the *unlock* operation can finish this operation and continue executing threads without needing to wait for any concurrent *lock* operations to finish (and vice versa). As a result, the mutex primitive in LFTHREADS tolerates arbitrary delays and even stop failures inside mutex operations without affecting the other processors' ability to do useful work, including performing operations on the same mutex.

The details of the *responsibility hand-off* method are given in the description of the operations, below:

The *lock* operation: Line L1 atomically increases the count of threads that want to access the mutex using Fetch-And-Add. If the old value was 0 the mutex was free and is now locked by the thread. Otherwise the mutex is likely to be locked and the current thread has to block. Line L3 stores the context of the current thread in its TCB and line L4 enqueues the TCB on the mutex's waiting queue. From now on, this invocation of *lock* is not associated with any thread.

However, the processor cannot just leave and do something else yet, because the thread that owned the mutex might have unlocked it (since line L1); this is checked by line L6 to L8. If the token read from m.hand-off is not null then an *unlock* has tried to unlock the mutex but found (at line U2) that although there is a thread waiting to lock the mutex, it has not yet appeared in the waiting queue (line H2). Therefore, the *unlock* has set the hand-off flag (line H5). However, it is possible that the hand-off flag was set after the thread enqueued by this *lock* (at line L4) had been serviced. Therefore, this processor should only attempt to take responsibility of the mutex if there is a thread available in the waiting queue. This is ensured by the *is_empty* test at line L7 and the CAS at line L8 which only succeeds if no other processor has taken responsibility of the mutex since line L6. If the CAS at line L8 succeeds, *lock* is now responsible for the mutex again and must find the thread wanting to lock the mutex. That thread (it might not be the same as the one enqueued by this *lock*) is dequeued from the waiting queue and this processor will proceed to execute it (line L9 - L10).

If the conditions at line L7 are not met or the CAS at line L8 is unsuccessful, the mutex is busy and the processor can safely leave to do other work (line L11).

To avoid ABA-problems (i.e. cases where CAS succeeds although the variable has been modified from its old value A to some value B and back to A) m.hand-off should, in addition to the processor id, include a per-processor sequence number. This is a well-known method in the literature, easy to implement and has been excluded from the presented code to make the presentation clearer.

The *trylock* operation: The operation will lock the mutex and return true if the mutex was unlocked. Otherwise it does nothing and returns false. The operation tries to lock the mutex by increasing the waiting count at line TL1. This will only succeed if the mutex was unlocked and there were no ongoing *lock* operations. If there are ongoing *lock* operations or some thread has locked the mutex, *trylock* will attempt to acquire the hand-off flag. This might succeed if the thread owning the mutex is trying to unlock it and did not find any thread in the waiting queue despite at least one ongoing *lock* operation. If the *trylock* operation succeeds in acquiring the hand-off flag it becomes the owner of the mutex and increases the waiting count at line TL3 before returning true. Otherwise *trylock* returns false.

The *unlock* operation: If there are no waiting threads *unlock* unlocks the mutex. Otherwise one of the waiting threads is made owner of the mutex and enqueued on the Ready_Queue. The operation begins by decreasing the waiting count at line U1, which was increased by this thread's call to *lock* or *trylock*. If the count becomes 0, there are no waiting threads and the *unlock* operation is done. Otherwise, there are at least one thread wanting to acquire the mutex and the *do_hand-off* procedure is used in order to either find the thread or hand-off the responsibility for the mutex.

If the waiting thread has been enqueued in the waiting queue, it is dequeued (line H2) and moved to the Ready_Queue (line H3) which completes the *unlock* operation. Otherwise, the waiting queue is empty and the *unlock* operation initiates a *responsibility hand-off* to get rid

of the responsibility for the mutex (line H5):

- The responsibility hand-off is successful and terminates if: (i) the waiting queue is still empty at line H6; in that case either the offending thread has not yet been enqueued there (in which case, it has not yet checked for hand-offs) or it has in fact already been dequeued (in which case, some other processor took responsibility for the mutex); or if (ii) the attempt to retake the hand-off flag at line H8 fails, in which case, some other processor has taken responsibility for the mutex. After a successful hand-off the processor leaves the *unlock* procedure (line H7 and H9).
- If the hand-off is unsuccessful, i.e. the CAS at line H8 succeeds, this processor is yet again responsible for the mutex and must repeat the hand-off procedure. Note that when a hand-off is unsuccessful, at least some other concurrent *lock* operation made progress, namely by completing an enqueue on the waiting queue (otherwise this *unlock* would have completed at lines H6 - H7). Note further that since the CAS at line H8 succeeded, none of the concurrent *lock* operations have executed line L6-L8 since the hand-off began.

Fault-tolerance Regarding *processor failures*, the procedures enable the highest achievable level of fault-tolerance for a mutex. Note that even though a *processor failure* while the *unlock* is moving a thread from the m.waiting queue to the Ready_Queue (between line H2 and H3) could cause the loss of two threads (i.e. the current one and the one being moved), the system behaviour in this case is indistinguishable from the case when the processor fails before line H2. In both cases the thread owning the mutex has failed before releasing ownership. At all other points a *processor failure* can cause the loss of at most one thread, namely the one whose context is executing.

4 Correctness of the synchronization in LFTHREADS

To prove the correctness of the synchronization in the thread library we need to show that the mutex primitive has the desired semantics. We will first show that the mutex operations are lock-free and linearizable with respect to the processors and then that the lock-free mutex implementation satisfies the conditions for mutual exclusion with respect to the behaviour of the application threads.

First we define (i) some notation that will facilitate the presentation of the arguments and (ii) establish some lemmas that will be used later to prove the safety, liveness, fairness and atomicity properties of the algorithm. Due to space limitations all proofs are omitted. The interested reader can find them in [9].

Definition 1 *A thread's call to a blocking operation Op is said to be* completed *when the processor executing the call leaves the blocked thread and goes on to do something else (e.g. executing another thread). The call is said to have* returned *when the thread (after becoming unblocked) continues its execution from the point of the call to Op.*

Definition 2 *A mutex m is* locked *when m.count* > 0 *and m.hand-off* $= null$*. Otherwise it is* unlocked*.*

Definition 3 *When a thread* τ*'s call to lock on a mutex m returns we say that thread* τ *has* locked *or* acquired *the mutex m. Similarly, we say that thread* τ *has* locked *or* acquired *the mutex m when the thread's call to trylock on the mutex m returns* $True$*.*

Further, when a thread τ *has acquired a mutex m by a lock or successful trylock operation and not yet released it by calling unlock we say that the thread* τ *is the* owner *of the mutex m (or that* τ owns *m).*

Lemma 1 *The value of the m.count variable is never negative and always greater than zero when a thread* owns *the mutex m.*

Lemma 2 *If m.hand-off* $\neq null$ *then m.count* > 0*.*

4.1 Lock-freedom

The lock-free property of the thread library operations will be established with respect to the processors. An operation is lock-free if it is guaranteed to complete in a bounded number of steps unless it is interfered with an unbounded number of times by other operations and every time operations interfere, at least one of them is guaranteed to make progress towards completion.

Theorem 1 *The mutex operations lock, trylock and unlock are all lock-free.*

The lock-freedom of *trylock* and *unlock*, with respect to application threads, follows trivially from their lock-freedom with respect to processors, since there are no context switches in them. The operation *lock* it is clearly neither non-blocking nor lock-free with respect to application threads, since a thread calling *lock* on a locked mutex should be blocked.

4.2 Linearizability

Linearizability guarantees that the result of any concurrent execution of operations is identical to a sequential execution of the operations where each operation takes effect atomically at a single point in time (its *linearization point*, referred to as LP below) within its duration in the original concurrent execution.

Theorem 2 *Operation lock is linearizable.*

Theorem 3 *Operation trylock is linearizable.*

Theorem 4 *Operation unlock is linearizable.*

4.3 Mutual exclusion properties

The mutual exclusion properties of the new mutex protocol are established with respect to application threads.

Theorem 5 (Safety) *For any mutex m and at any time t there is at most one thread τ such that τ is the owner of m at time t.*

Lemma 3 *No thread is left blocked in the waiting queue of an unlocked mutex m when all concurrent operations concerning m have completed.*

Lemma 4 *A mutex is locked if and only if it is* owned *by a thread.*

Lemma 5 *A thread τ waiting to acquire a mutex m in a call to lock will at most have to wait for the thread currently owning m and all threads that have called lock on m before τ's call to lock enqueued τ on the m.waiting queue.*

Theorem 6 (Liveness I) *A thread τ waiting to acquire a mutex m will eventually acquire the mutex once its lock operation has enqueued τ on the m.waiting queue.*

Theorem 7 (Liveness II) *A thread τ wanting to acquire a mutex m can only be starved if there is an unbounded number of lock operations on m performed by threads on other processors.*

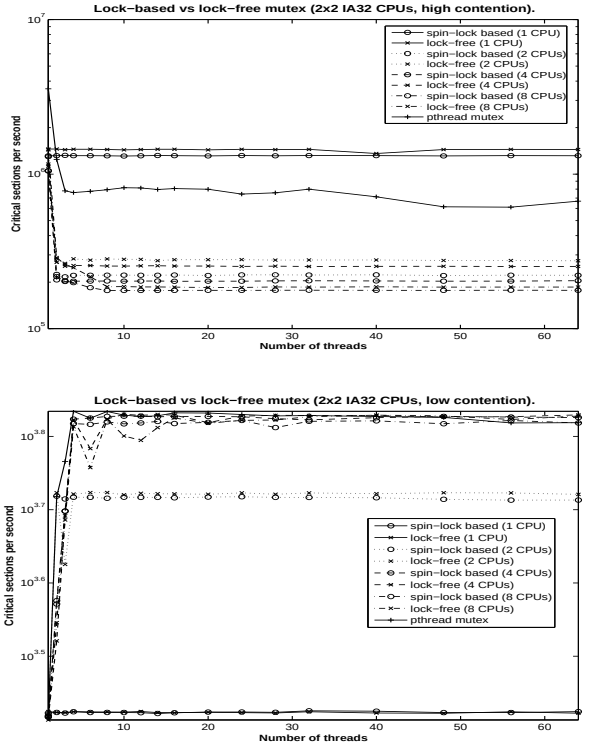
Theorem 8 (Fairness) *A thread τ wanting to acquire a mutex m will only have to wait for the threads whose lock operation enqueued them on the m.waiting queue before τ was enqueued there.*

5 Experimental study

The primary contribution of this work is to enhance qualitative properties of thread library operations, such as the tolerance to delays and processor failures. However, since lock-freedom may also imply performance/scalability benefits with increasing number of processors, we also wanted to observe this aspect of the impact of the lock-free mutex implementation. We made an implementation of the mutex object and the thread operations on the GNU/Linux operating system. The implementation is written in the C programming language and was done entirely at the user-level using "cloned"[e] processes as *virtual processors* for running the threads. The implementation uses the lock-free queue in [34] for the mutex waiting queue and the Ready_Queue. To ensure sufficient memory consistency for synchronization variables, memory barriers surround all CAS and FAA instructions and the writes at lines L6 and H5. The lock-based mutex object implementation uses a test and test-and-set type spin-lock to protect the mutex state. Unlike the use of spin-locks in an OS kernel, where usually neither preemptions nor interrupts are allowed while holding a spin-lock, our virtual processors can be interrupted by the OS kernel due to such events. This behaviour matches the asynchronous processors in our system model well.

[e] "Cloned" processes share the same address space, file descriptor table and signal handlers etc and are also the basis of Linux's native pthread library implementation.

Figure 5 Mutex performance in LFTHREADS and pthreads at high (top) and low (bottom) contention.



The experiments were run on a PC with two Intel Xeon 2.80GHz processors (acting as 4 due to hyper-threading) using the GNU/Linux operating system with kernel version 2.6.9. The microbenchmark used for the experimental evaluation consists of a single critical section protected by a mutex and a set of threads that each try to enter the critical section a fixed number of times. The contention level on the mutex was controlled by changing the amount of work done outside the critical section.

We evaluated the following thread library configurations experimentally:

- The lock-free mutex using the protocol presented in

this paper, using 1, 2, 4 and 8 virtual processors to run the threads.
- The spin-lock based mutex, using 1, 2, 4 and 8 virtual processors to run the threads.
- The platform's standard pthreads library and a standard pthread mutex. The pthreads library on GNU/Linux use kernel-level "cloned" processes as threads, which are scheduled on all available processors, i.e. the pthreads are at the same level as the virtual processors in LFTHREADS. This difference in scheduling makes it difficult to interpret the pthreads results with respect to the others; i.e. the pthreads results should be considered to be primarily for reference.

Each test configuration was run 10 times. The diagrams present the mean of these 10 runs.

High contention In Figure 5 (top) we show the microbenchmark results when all work is done inside the critical section, that is, the contention on the mutex is high. In this case the desired result would be that the throughput, i.e. the number of critical sections executed per second, for an implementation stays the same regardless of the number of threads or virtual (processors). This should imply that the synchronization scales well. However, in reality the throughput decreases with increasing number of virtual processors, mainly due to preemptions inside the critical section (but for spin-locks also inside mutex operations) and synchronization overhead. Further, going from a single processor to more than one processor for our thread library implies a cost since with more than one processor the thread contexts will have to be stored and restored much more often due to threads being blocked on the mutex. (Note that threads currently use non-preemptive scheduling in our implementation so with only one virtual processor the threads will run to completion one after the other without any extra blocking.) The results indicate that the lock-free mutex has less overhead than the lock-based one in similar configurations.

Low contention In Figure 5 (bottom) we show the results from a microbenchmark where the threads perform 1000 times more work outside the critical section than inside, making the contention on the mutex low. With the majority of the work outside the critical section, the expected behaviour is a linear throughput increase over threads until all (physical) processors are in use by threads, thereafter constant throughput as the processors are saturated with threads running outside the critical section. The results agrees with the expected behaviour; we see that from one to two virtual processors the throughput doubles in both the lock-free and spin-lock based cases. (Recall that the latter is a test-and-test-and-set-based implementation, which is favoured under low contention). Note that the step to 4 virtual processors does not double the throughput — this is due to hyper-threading, there are not 4 physical processors available. Similar behaviour can also be seen in the pthread-based case. Further, the lock-free mutex shows similar or higher throughput than the spin-lock-based one for the same number of virtual processors; it also shows comparable and even better performance than the pthread-based case when the number of threads is large and there are "enough" virtual processors (i.e. more than the physical processors).

Summarizing, we observe that the LFTHREADS thread library's lock-free mutex protocol implies comparable or better throughput than the lock-(test-and-test-and-set-)based implementation, both in high- and in low-contention scenaria for the same number of virtual processors, besides offering the qualitative advantages in tolerance against slow, delayed or crashed threads, as discussed earlier in the paper.

6 Conclusions

In this paper we have presented the LFTHREADS library and the first lock-free implementation of a blocking synchronization primitive; as part of this contribution we have introduced the responsibility hand-off (RHO) synchronization method. Besides supporting a thread-library interface with fault-tolerance properties, we regard the RHO method as a conceptual contribution, which can be useful in multiprocessors and multicore systems in general.

We have implemented the LFTHREADS library on a PC multiprocessor platform with two Intel Xeon processors running the GNU/Linux operating system and using processes as virtual processors. The implementation does not need any modifications to the operating system kernel. Although our present implementation is entirely at the user-level, its algorithms are well suited for use also in a kernel - user-level divided setting. With our method a significant benefit would be that there is no need for spin-locks and/or disabling interrupts in either the user-level or the kernel-level part.

Our implementation constitutes a proof-of-concept of the lock-free implementation of the blocking primitive introduced in the paper and serves as basis for an experimental study of its performance. The experimental study presented here, using a mutex-intensive microbenchmark, shows positive performance figures. Moreover, this implementation can also serve as basis for further development, for porting the library to other multiprocessors and experimenting with parallel applications such as the Spark98 matrix kernels or applications from the SPLASH-2 suite.

References

[1] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *ACM Trans. on Computer Systems*, pages 53–79, 1992.

[3] G. Barnes. A method for implementing lock-free shared data structures. In *Proc. of the 5th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 261–270. SIGACT and SIGARCH, 1993. Extended abstract.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing,. In *Proc. of the 35th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[5] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global edf scheduling on multiprocessors. In *Proc. of the 18th Euromicro Conf. on Real-Time Systems*, pages 75–84. IEEE Computer Society, 2006.

[6] M. J. Feeley, J. S. Chase, and E. D. Lazowska. User-level threads and interprocess communication. Technical Report TR-93-02-03, University of Washington, Department of Computer Science and Engineering, 1993.

[7] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proc. of the Ottawa Linux Symp.*, pages 479–494, 2002.

[8] A. Gidenstam and M. Papatriantafilou. LFthreads: A lock-free thread library. In *Proc. of the 11th Int. Conf. on Principles of Distributed Systems (OPODIS)*, pages 217 – 231. Springer, 2007.

[9] A. Gidenstam and M. Papatriantafilou. LFthreads: A lock-free thread library. Technical Report MPI-I-2007-1-003, Max-Planck-Institut für Informatik, Algorithms and Complexity, 2007.

[10] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Practical and efficient lock-free garbage collection based on reference counting. In *Proc. of the 8th Int. Symp. on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 202 – 207. IEEE Computer Society, 2005.

[11] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Allocating memory in a lock-free manner. In *Proc. of the 13th Annual European Symp. on Algorithms (ESA)*, pages 329 – 242. Springer Verlag, 2005.

[12] M. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[13] M. B. Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, 1999.

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programming Languages and Systems*, 15(5):745–770, 1993.

[15] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. on Computer Systems*, 23(2):146–196, 2005.

[16] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd Int. Conf. on Distributed Computing Systems (ICDCS)*, page 522. IEEE Computer Society, 2003.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[18] P. Holman and J. H. Anderson. Locking under pfair scheduling. *ACM Trans. Computer Systems*, 24(2):140–174, 2006.

[19] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proc. of the 12th Int. Symp. on Distributed Computing (DISC)*, pages 216–230. Springer Verlag, 1998.

[20] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Trans. Computer Systems*, 15(1):3–40, 1997.

[21] L. Lamport. On interprocess communication–part i: Basic formalism, part ii: Algorithms. *Distributed Computing*, 1:77–101, 1986.

[22] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the 19th Int. Conf. on Distributed Systems (DISC)*, pages 354–368. Springer, 2005.

[23] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[24] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, 1991.

[25] M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of SIGPLAN 2004 Conf. on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, 2004.

[26] M. M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, 1995.

[27] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. of the 16th annual ACM Symp. on Principles of Distributed Computing*, pages 219–228, 1997.

[28] H. Oguma and Y. Nakayama. A scheduling mechanism for lock-free operation of a lightweight process library for SMP computers. In *Proc. of the 8th Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 235–242, 2001.

[29] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 204–213. ACM Press, 1995.

[30] Multithreading in the solaris operating environment. Technical report, Sun Microsystems, 2002.

[31] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.

[32] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. In *Proc. of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*. Springer Verlag, 2002.

[33] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In *Proc. of the ACM SIGMETRICS 2001/Performance 2001*, pages 320–321. ACM press, 2001.

[34] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proc. 13th ACM Symp. on Parallel Algorithms and Architectures*, pages 134–143. ACM Press, 2001.

[35] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proc. of the 11th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 212–222. ACM Press, 1992.

[36] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 214–222. ACM, 1995.

[37] J. Zahorjan, E. D. Lazowska, and D. L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel processors. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):180–198, 1991.

[38] K. M. Zuberi and K. G. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proc. of the 3rd IEEE Real-Time Technology and Applications Symp. (RTAS)*, pages 25–37. IEEE, 1997.

Wool-A Work Stealing Library

Karl-Filip Faxén
Swedish Institute of Computer Science
kff@sics.se

Abstract

This paper presents some preliminary results on a small light weight user level task management library called Wool. The Wool task scheduler is based on work stealing. The objective of the library is to provide a reasonably convenient programming interface (in particular by not forcing the programmer to write in continuation passing style) in ordinary C while still having a very low task creation overhead. Several task scheduling systems based on work stealing exists, but they are typically either programming languages like Cilk-5 or based on C++ like the Intel TBB or C# as in the Microsoft TPL. Our main conclusions are that such a direct style interface is indeed possible and yields performance that is comparable to that of the Intel TBB.

1 Introduction

A task management library supports *task parallel programming*, a popular and promising approach to exploiting the performance potential of multi(core) processors. Tasks are light weight units of work. In our model, tasks synchronize only when a task is created or completes execution; in pthreads terminology we support only the operations create, exit and join. These limited forms of synchronization makes it particularly important that the overheads associated with tasks are low, ideally on the order of the cost of a procedure call.

In a work stealing scheduler, each worker thread maintains a pool of tasks that are ready to execute. Tasks are created dynamically and are added to the pool of the creating worker. A worker executes tasks form its own pool if possible, but if the pool of a worker is empty, the worker steals a task from the task pool of a randomly chosen worker (the *victim*). Task stealing is efficient since it is the processors that have nothing useful to do that do most of the load balancing.

2 Programming with Wool

The Wool API is similar to (and inspired by) that of Cilk-5 [2] and consists of constructs for defining, spawning and waiting for the completion of tasks. Just like Cilk-5, and in contrast to for instance the Intel TBB, the Wool API is direct style, facilitating the conversion of existing code. In a direct style API, there is a synchronization operation (called `SYNC` in Wool) which syntactically looks like a procedure call. In contrast, a continuation passing API specifies a task to spawn when a set of other tasks have completed. Thus if the programmer wishes to first do some compuations in parallel, then (when the earlier computations have completed) do another computation, that can be accomplished by placing a `SYNC` in between. In a continuation passing API, the programmer instead indicates a new task to spawn when the earlier tasks have completed.

Figure 1 shows a naive implementation of the Fibbonacci function in Wool. A sequential (but still naive) function (`seqfib`) is used for small arguments.

Wool provides a family of task definition macros of the form

- `TASK_`*`n`*`(`*`rtype`*`, `*`name`*`, `*`aty_1`*`, `*`arg_1`*`, ..., `*`aty_n`*`, `*`arg_n`*`)` which define tasks with return type *`rtype`*, name *`name`*, and arguments with their corresponding types.

Given such a definition of a task called *`name`*, Wool provides the following operations:

- `SPAWN(`*`name`*`, `*`arg_1`*`, ..., `*`arg_n`*`)` which spawns the task to make it available for parallel execution. A spawn has no return value (that is, it has return type `void`)
- `SYNC(`*`name`*`)` which synchronizes with the most recently spawned task that has not been synchronized. A sync returns the value returned by the task.


```
#include <stdio.h>
#include <stdlib.h>
#include "wool.h"

int threshold;

int seqfib( int n )
{
  if( n<2 ) {
    return n;
  } else {
    return seqfib( n-1 )
         + seqfib( n-2 );
  }
}

TASK_1( int, fib, int, n )
{
  if( n<threshold )
    return seqfib( n );
  else {
    int a,b;
    SPAWN( fib, n-2 );
    a = CALL( fib, n-1 );
    b = SYNC( fib );
    return a+b;
  }
}

TASK_2( int, main, int, argc,
                   char **, argv )
{
  int n;

  threshold = atoi( argv[1] );
  n = CALL( fib, atoi( argv[2] ) );
  printf( "%d\n", n );
}
```

Figure 1. A simple Fibbonacci function

- `CALL(`*name*`, `*arg_1*`, ..., `*arg_n*`)` which invokes the task as a function call, semantically equivalent to a spawn immediately followed by a sync. A call returns the return value of the task.

The `CALL` operation is avalable as an optimization, saving the synchronization overhead associated with `SPAWN` and `SYNC`.

Compared to Cilk-5, which is implemented in a compiler, the Wool API is slightly coarser. A Cilk-5 `sync` synchronizes with all tasks started since the beginning of the current function call, and the `spawn` operation returns the value of the task (although that value can not be used before the next `sync`).

We are planning a few extensions to the API, in particular parallel for-loops (so called DOALL loops) similar to those found in openMP and the Intel TBB. The syntax of such a construct is approximately

- `FOR(`*name*`, `*from*`, `*to*`, `*arg_2*`, ..., `*arg_n*`)`

where *name* is a task whose first argument is of type `int`, *from* and *to* are the loop bounds and the *arg_i* are the rest of the arguments to the task.

3 The implementation of Wool

Wool allows the programmer to express all parallelism available in the problem; as much of that parallelism as is needed to keep the particular machine executing the program busy will be realized. The excess tasks are executed sequentially, called *inlining* the task, basically as procedure calls (by far the most common case). Spawning a task makes it available for parallel execution, but does not guarantee that it will actually be executed in parallel. If a processor becomes idle it tries to steal spawned but not yet executed tasks from some other processor, and it is only such stolen tasks that execute in parallel with other tasks spwned on the same processor.

One of the main objectives of Wool is to investigate how fast spawning can be made; if a spawn is cheap, the programmer does not need to choose whether or not to use parallelism but can always spawn independent computations and let the implementation worry about the overheads. We also want to stay withing a direct style execution model as opposed to a continuation passing model.

For many programs, spawning is the dominant source of overhead. This happens when spawning is much more frequent than stealing, as is the case in fine grained recursive divide and conquer programs. Thus the first order of business when implementing Wool was to ensure that spawning a task and synchronizing with

an unstolen task is as cheap as possible. This brings us to a brief discussion of the basic data structures used by the Wool scheduler.

In Wool, each worker has a small data structure called a *worker descriptor* which contains the administrative data structures for that worker. In particular it contains:

- An array of task descriptors for tasks spawned by this worker.
- A pointer to the next task to steal, used by thieves, called `bot`.
- A lock that is mostly used to achieve mutual exclusion between thieves, but that is also used as a fall back in the synchronization of thief and victim.
- Various other things, including statistics counters.

Conspicuously absent is the `top` pointer used by worker to access its own task queue; this pointer is a hidden extra parameter to each task and its initial value is passed by the run time system to the `main` task. Technically, `top` is a piece of thread local data; it is different for each task and can thus not be a global variable. Pthreads provides functions for dealing with thread local data (`pthread_setspecific`, `pthread_getspecific`, ...) but these are to expensive to use for every spawn, hence we pass it around instead. In a compiled implementation (where the code generator is tailored to Wool), `top` could have a reserved register that did not need to be saved accross task and procedure invocations.

The pointer to the worker descriptor, `self`, is however used less often and is currently managed using Pthreads thread specific data functions.

The astute reader may by now be wondering how a thief and its victim synchronize since the classic way to do so in work stealing is to have both `top` and `bot` in the worker descriptor and have worker and thief updating and comparing them. We have not used this design, since it will cause a unnecessry coherence transactions when workers and thiefs access the worker descriptor and since it is incompatible with keeping `top` in a register. Instead, thieves and victims synchronize through the task descriptors in the task queue of the victim, to which we now turn our attention.

A task descriptor contains the following information:

- A pointer `f` to the code (C-function) that implements the task.
- A pointer `balarm` indicating whether the task has been stolen.
- The arguments the task was spawned with.

```
int steal( Worker *victim )
{
   lock( victim->lck );
   t = victim->bot;
   t->balarm = STOLEN;
   memory_barrier();
   if( t->f == INLINED ) {
      return 0;
      unlock( victim->lck );
   } else {
      victim->bot++;
      unlock( victim->lck )
       ...     // Run the task
   }
}

sync( Task *t )
{
   t->f = INLINED;
   memory_barrier();
   if( t->balarm = STOLEN ) {
      lock( self->lck );
      if( t->balarm == STOLEN ) {
        unlock( self->lck );
        ...
      } else {
         unlock( self->lck );
         ...
      }
   }
}
```

Figure 2. Synchronization between thief and victim

- The return value of the task (shares space with the arguments).

We use the `f` and `balarm` fields in a THE style protocol where a thief indicates interest in the task by setting `balarm` to a specific value while the owner of the queue signals synchronizing with the task by writing to the `f` field. Consider the simplified code in Figure 2: Both thief and victim first do their writes, then check if the other party has written. The thief always acquires the lock of the victim, both to ensure that there is only one thief at a time and to resolve the possible deadlock when both finds the others field written. In that case the thief backs off while the victim acquires the lock and then rechecks `balarm`, at which point it can tell

whether the thief had just arrived at the same time or earlier. The memory barriers are necessary on modern architectures with relaxed memory consistency models.

3.1 Waiting

This scheme raises the issue of what to do when a sync finds that the task has been stolen. The sync can not complete until the task is finished, so some form of wait is needed. This easily leads to a considerable loss in performance since the worker executing the sync becomes idle. One possibility would be for that worker (call it A) to steal some work; however, there is a subtle issue involved in that idea. If A starts executing other tasks, and the worker that stole the original task (B) completes, A can not immediately continue from the blocked sync until the new tasks complete. This is problematic since the work after the blocked sync is not in a form that is stealable (this is exactly a consequence of not forcing the programmer to use a continuation passing style). Thus we have work in the system that is logically ready to execute but that no worker can run.

3.1.1 Leapfrogging

In the leapfrogging tehnique [3], a worker A that needs to wait at a sync becuase the task to sync with is stolen can still find some other work to do, while avoiding the problem mentioned above, by stealing only from B. The tasks in the pool of B must have been generated while processing the task stolen from A, and must complete before that task. Thus ut is not possible that the sync that A (logically) waits for becomes unblocked while A processes new tasks.

3.1.2 Parking

In Wool we have implemented a different scheme. The idea is to rely on modern threading implementations to reasonably efficient and to use more workers than processors (cores). Thus a worker that finds a sync actually does wait, but the thread scheduler (typically in the kernel) can then run another thread. However, this entails several workers being time shared on a single processor (core) which typically leads to somewhat higher overheads and, more importantly, worse locality and more cache misses. To deal with this problem we use a novel technique we call *parking*. The idea is that most of the time, only as many workers as there are cores are actively executing tasks and the rest are blocked. When a worker needs to wait at a sync, it unblocks one of the parked workers and then goes to sleep waiting for the sync'ed task. When it completes, there will for a while be more workers than cores, but as soon as some worker finds its task pool empty, it checks the number of active workers and parks itself.

4 Experimental results

We have used three microbenchmarks to characterize the performance and overheads of Wool. These are `fib`, discussed above, `qs`, a recursive divide and conquer implementation of the quicksort algorithm, and a synthetic program called `stress` where we can control the amount and granularity of parallelism precisely (thus being able to "stress test" the implementation). The `fib` and `qs` benchmarks also have cutoff parameters below which a sequential algorithm is used. For `fib` it is the argument and for `qs` the number of elements in the array.

Since the machines we have used vary greatly in performance, we have run our benchmarks with different inputs, as shown in Table 2 together with the sequential timings. For these runs, input parameters giving negligible overhead for parallelism were used. The older Sun machine (`scheutz`) has hardware problems which gives the processors different performance; the sequential time is an average of ten runs.

We have evaluated our three micro benchmarks on the three different systems using five variations on the Wool implementation of waiting and stealing.

wait A worker that tries to sync with a stolen task simply goes to sleep on a condition variable that is signalled by the thief when finished with the task.

park Uses parking with total threads either two or three times the number of active threads.

leap Uses leapfrogging, that is, a worker that needs to wait for a thief spends its time stealing tasks from that same thief.

back Leapfrogging plus exponential back off, both for unsuccessful leaps and steals. Back off starts at 4000 iterations of a trivial loop (the same as in the stress program) for steals and 1000 iterations for stealing.

The speedups we report are wall clock times measured using the Unix `time` command and is the average of five runs. They are relative to the parallel version of the code running on one processor, but while this is slower than the best sequential execution for qs and the fine grained regime of stress, we believe they can be further reduced although we have no yet implemented our ideas for these optimizations.

Model	Name	CPU	CPUs	Cores	Threads	Clock	OS
Sun Enterprise	`scheutz`	Sun UltraSPARC II	8	8	8	248 MHz	Solaris
Sun Fire T1000	`millennium`	UltraSPARC T1	1	6	24	1 GHz	Solaris
Mac	`small`	Intel Core 2 Quad	2	8	8	2.8 GHz	MacOS

Table 1. Machine characteristics

	fib		qs		stress	
	Input	Time	Input	Time	Input	Time
`scheutz`	37	5.91	2M	4.51	240M	1.96
`millennium`	39	11.4	4M	6.78	670M	4.03
`small`	42	2.16	20M	3.25	2G	1.24

Table 2. Inputs and sequential run times on the different systems

4.1 fib

The fib program (given in figure 1) is a very simple recursive divide and conquer program which does not load the memory system very heavily as it only references a couple of global variables, the stack and the Wool administrative data structures, which are small and/or have a very localized reference pattern. Thus it measures the CPU overhead together with the unavoidable synchronization costs.

Unsurprisingly, `small` and `scheutz` achieve very good speed up, but for `millennium` performance levels off after a few workers. This is natural since the processor only has six cores, each with four threads. This design shows its strength when there are many cache misses or other kinds of stalls, but presumably fib creates respectable utilization of the core with a single active thread.

One pattern that we see, will continue to see, is that **wait** performs worse than the other alternatives. This effect is due to load imbalance, which can be seen by relating cpu time, elapsed time and number of processors.

4.2 qs

This is another recursive divide and conquer program, but in contrast to fib, it makes heavy use of the memory system. This is expecially true for `millennium` for which performance even starts to drop for large numbers of workers. Here is another interesting phenomenon: The cores in the processor share a common second level cache, and the threads in a core even share the first level cache. This means that, as the number of workers increase, cache miss rates also rise steeply to the point where memory bandwidth becomes the limiting factor. The eight core `small` fares best with a speedup of over five on all eight cores.

4.3 stress

The stress program is a synthetic benchmark designed to be easy to vary the grainsize of. There is a sequential outer loop that for each iteration starts a parallel, perfectly balances divide and conquer computation of a given depth. At the leaves of this tree, a simple loop is iterated. Thus for n iterations of the outer loop, depth d of the tree and m iteration of the leaf loop, the program does $n \times 2^d \times m$ iterations of the innermost loop. Each parallel region is $n \times 2^d$ innermost iterations.

Table 3 shows timings in nanoseconds and cycles for single iterations of the innermost loop in stress and for the cost of uncontended spawn and sync operations (that is, times with only a single worker). It also gives the sequential execution time in microseconds and clock cycles for each parallel region in the program (one iteration of the outermost loop is a parallel region). In a recent paper [1], programmers using the Intel TBB are advised to try to create task granularities on the order of 100k cycles. If one compares this with Figures 5 and 6 one sees that when the parallel region is on the order of 1300k cycles, both `scheutz` and `small` manage to reach quite respectable speedups (around 6 on 7 cores). Note that our parallel regions are not the same as tasks; the parallel region discussed above contains *at least* six tasks, so we are within a factor of two of the above mentioned advice. In fact, our parallel region is made up of about 2^{17}small tasks. With the larger grain size and a parallel region of around 8400–8600k cycles, the speed ups are even better.

5 Conclusions

Wool achieves an low spawning overhead and reasonable stealing cost while providing a direct style API.

Machine	time/iteration		time / spawn-sync		time / parallel region			
					m=20, d=17		m=200, d=17	
	ns	cycles	ns	cycles	us	Kcycles	us	Kcycles
`scheutz`	8.2	2	242	60	5321	1310	34184	8388
`millennium`	6	6	113	113	3054	3054	21653	21653
`small`	0.71	2	23	65	485	1376	3041	8651

Table 3. Iteration times, uncontended overheads and parallel region sizes for stress

Much work remains, though, before the goal of extremely low overhead task creation and stealing is realized. In particular, batching synchronization (memory barriers) in spawn and sync is necessary to get their overheads close to that of a procedure call. Stealing could also be improved by avoiding the use of Pthreads locks and instead use more low level synchronization instructions as well as by moving to a more adaptive back off policy.

6 Acknowledgements

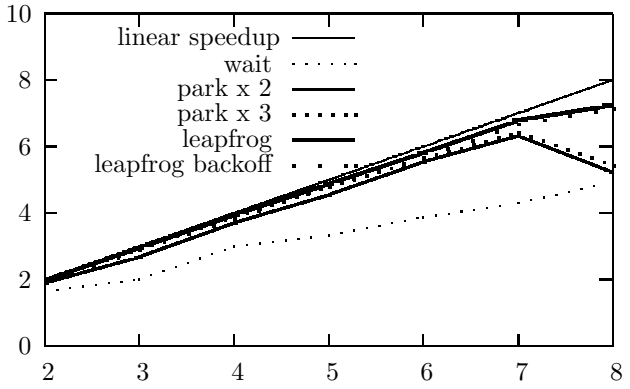
Cosmin Arad graciously made `small` available for the experimental work.

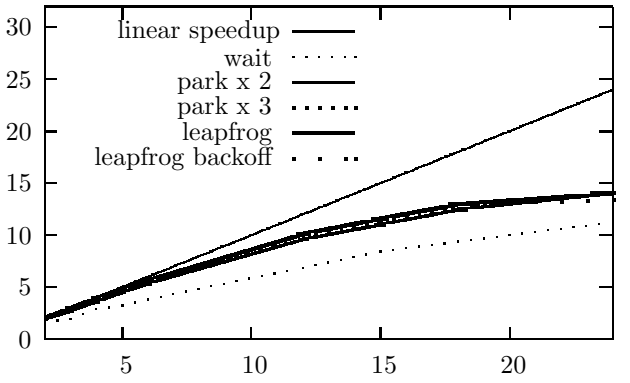
References

[1] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of the intel threading building blocks runtime system. In *International Symposium on Workload Characterization (IISWC 2008)*, August 2008.

[2] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[3] David B. Wagner and Bradley G. Calder. Leapfrogging: a portable technique for implementing efficient futures. *SIGPLAN Not.*, 28(7):208–217, 1993.





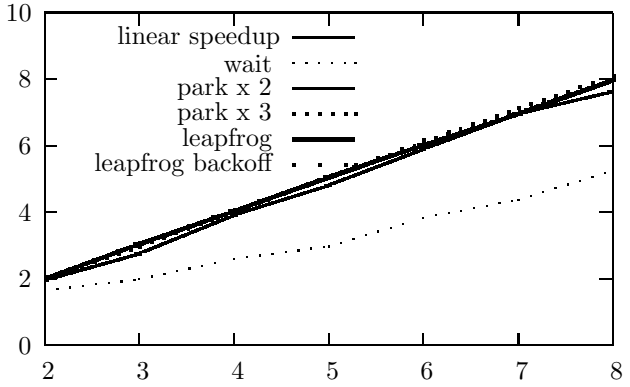
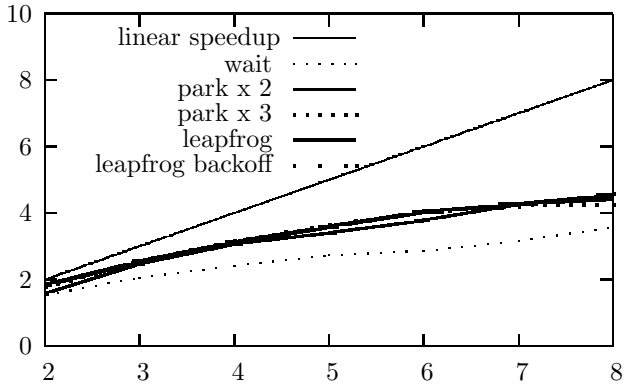
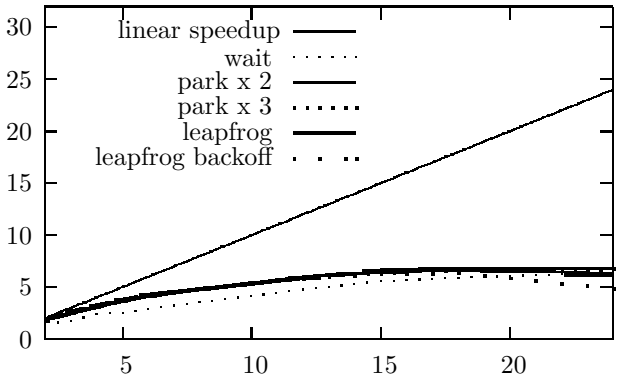


Figure 3. Speedups for fib (threshold 4) on (top to bottom) `scheutz`**,** `millennium` **and** `small`





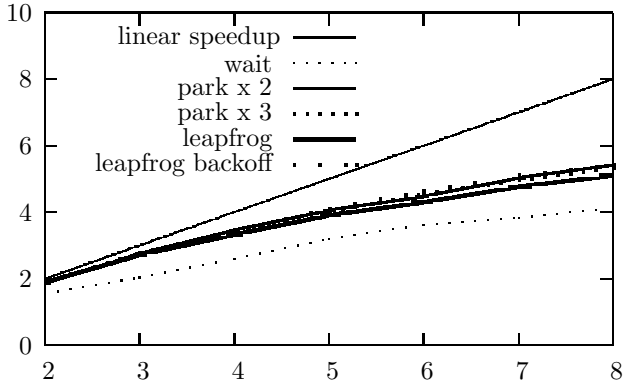
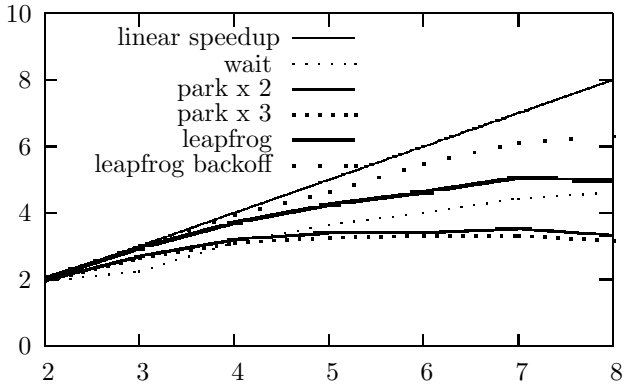
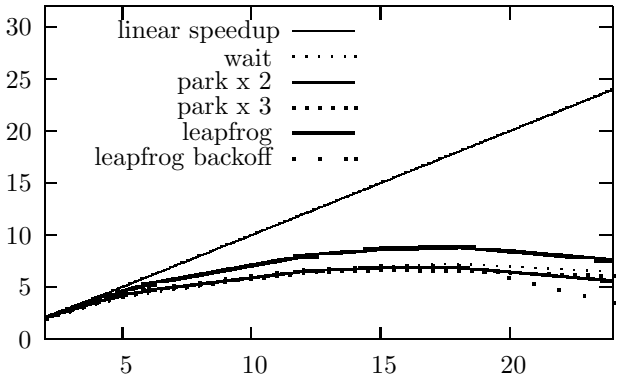


Figure 4. Speedups for qs (threshold 2) on (top to bottom) `scheutz`**,** `millennium` **and** `small`





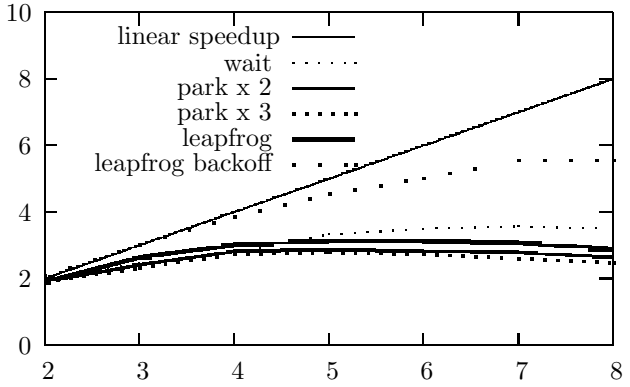
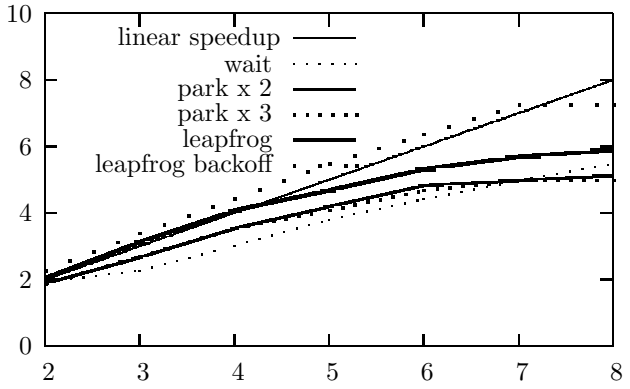
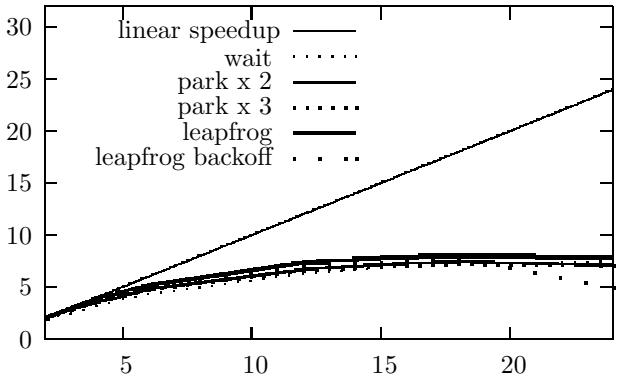


Figure 5. Speedups for stress on (top to bottom) `scheutz`**,** `millennium` **and** `small` **for** $n = 20$ **and** $d = 17$





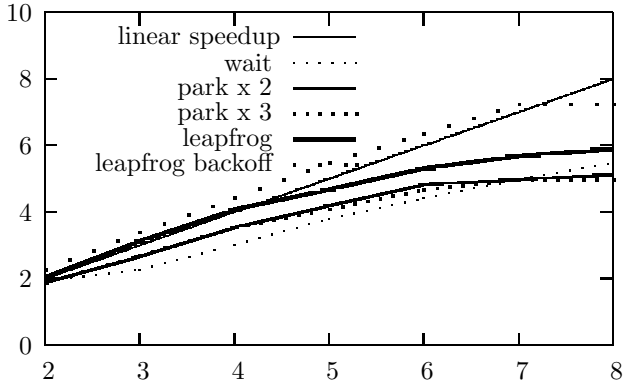


Figure 6. Speedups for stress on (top to bottom) `scheutz`**,** `millennium` **and** `small` **for** $n = 200$ **and** $d = 17$