Proceedings

5th Workshop on Optimizations for DSP and Embedded Systems (ODES-5)

March 11, 2007
San Jose, California

In conjunction with:
International Symposium on Code Generation and Optimization (CGO)

Sponsored by:
IEEE Computer Society and ACM SIGMICRO
in cooperation with ACM SIGPLAN

ACKNOWLEDGMENTS

The organizers would like to thank all the people who made this year's *Workshop for Optimization of DPS and Embedded Systems* possible:

- The authors who submitted a paper. Thanks to them we were able to select 9 good quality papers for this year's program.
- The program committee as well as Praveen Raghavan, Bart Mesman and Amir Hossein Ghamarian for their excellent reviews. As a submitter to previous ODES versions I can truly say ODES reviews always have been and still are of a very high quality.
- Edward Lee and Roberto Costa for accepting to give the keynote and an invited talk respectively, both on a very interesting topic for the ODES audience.
- The CGO organizers, especially David Tarditi, for taking care of all the logistics and for hosting the workshop.

We hope you have an interesting workshop,

Deepu Talla (deepu@ti.com),
Tom Vander Aa (vanderaa@imec.be),

Program Committee

Francisco Barat	NXP
Shuvra Bhattacharrya	University of Maryland
Francois Bodin	IRISA
John Cavazos	University of Edinburgh
Henk Corporaal	Technical University Eindhoven
Heiko Falk	University of Dortmund
Jose Fridman	Analog Devices
Murali Jayapala	IMEC
Tor Jeremiassen	Texas Instruments
Ossi Kalevo	Nokia
Trevor Mudge	University of Michigan
Vijay Narayanan	Penn State University
Scott Rixner	Rice University
Nat Seshan	Texas Instruments
Gary Tyson	Florida State University

PROGRAM

8:15 AM Introduction

8:30 AM Session 1: Memory

A Novel Scatter Search Procedure for Effective Memory Assignment for Dual-Bank DSPs
Gary Grewal (University of Guelph), S. Coros (University of British Columbia)

Use of an Embedded Configurable Memory for Stream Image Processing
Michael G. Benjamin and David Kaeli (Northeastern University)

An ILP Formulation for Recomputation Based SMP Management for Embedded CMPs
Hakduran Koc and Ehat Ercanli (Syracuse University), Mahmut T. Kandemir (The Pensylvania State University)

A Code Layout Framework for Embedded Processors with Configurable Memory Hierarchy
Kaushal Sanghai and David Kaeli (Northeastern University), Alex Raikman and Ken Butler (Analog Devices)

10:30 AM Break

11:00 AM Keynote

Challenges with Concurrency and Time in Embedded Software
Edward A. Lee (University of California at Berkeley)

12:00 PM Lunch (provided)

1:30 PM Session 2: Backend Compilers

Register Allocation for Processors with Dynamically Reconfigurable Register Banks
Ralf Dreesen, Michael Humann, Michael Thies and Uwe Kastens (University of Paderborn)

Compiler Optimization for the Cell Architecture

Junggyu Park, Alexander Kirnasov, Daehyun Cho, Byungchang Cha and Hyojung Song (Samsung Electronics)

Invited Talk: GCC-based CLI Toolchain for Media Processing
Roberto Costa, Andrea C. Ornstein, Erven Rohou and Andrea Bona (STMicroelectronics)

3:00 PM Break

3:30 PM Session 3: Architectures and Applications

Empirical Auto-tuning Code Generator for FFT and Trigonometric Transforms
Ayaz Ali and Lennart Johnsson (University of Houston, Texas), Dragan Mirkovic (The University of Texas, Houston)

Enabling WordWidth Aware Energy and Performance Optimizations for Embedded Processors
Andy Lambrechts, Praveen Raghavan, David Novo and Estela Rey Ramos (IMEC)

A Coprocessor Accelerator for Model Predictive Control
Panagiotis Vouzis and Mark Arnold (Lehigh University), Leonidas Bleris (Harvard University) Mayuresh Kothare (Lehigh University), Yongho Cha (Advanced Digital Chips)

A Novel Scatter Search Procedure for Effective Memory Assignment for Dual-Bank DSPs

G. Gréwal

Computing and Information Science
University of Guelph
Guelph, ON, Canada, N1L1N5

S. Coros

Computer Science
University of British Columbia
Vancouver, BC, Canada, V6T124

February 27, 2007

Abstract

To increase memory bandwidth, many programmable Digital Signal Processors (DSPs) employ two on-chip data memories. This architectural feature supports higher memory bandwidth by allowing multiple data memory accesses to occur in parallel. Exploiting dual memory banks, however, is a challenging problem for compilers. This, in part, is due to the instruction-level parallelism, small numbers of registers, and highly specialized register capabilities of most DSPs. In this paper, we present a new methodology based on *Scatter Search* for assigning data to dual-bank memories. Our approach is global, and integrates several important issues in memory assignment within a single model. Special effort is made to identify those data objects that could potentially benefit from an assignment to a specific memory, or perhaps duplication in both memories. Our computational results show that the Scatter Search is able to achieve a 54% reduction in the number of memory cycles and a reduction in the range of 7% to 42% in the total number of cycles when tested with well-known DSP kernels and applications.

1 Introduction

To increase memory bandwidth, many programmable Digital-Signal Processors (DSPs) employ two data memories. Examples include the Motorola DSP56000, SGS-Thomson 18950, Analog Devices ADSP2106x, and NEC mPD77016, just to name a few. The objective is to improve performance by maximizing the parallelism in the software pipeline. Parallelism is promoted by concurrently loading pairs of operands from opposite memories whenever an operation requires two memory-resident values. Assigning data in the best way to dual memories, however, remains an elusive problem.

Whenever operands must be fetched from the same memory, their load operations must occur on different control steps. Furthermore, it may not be possible to perform all of the required loads in parallel with the existing ALU operations. If even a few consecutive ALU operations pull too many operands from the same memory, additional instructions may be needed just to load values.

The register structure of the target architecture also has a direct effect on memory assignment. On some machines, the operand registers can only receive values from one corresponding memory; on others they are connected only to certain input ports of the ALU; on other machines, both restrictions apply. Such restrictions may occur only when the load operation is done in parallel with other operations, but parallelism is generally preferred for other reasons. The implication is that when a value appears in the "wrong" memory, it must first be fetched, then copied from the register associated with the memory to a register connected to the input terminal of the ALU. Non-commutative operations, like subtraction and division, may make a specific memory the preferred one, while (isolated) commutative operations only prefer that opposite memories be employed.

The problem can have additional constraints, which include: having certain objects pre-assigned to a specific memory; requiring certain objects to be either in the same memory or in opposite memories as certain others; and limiting the total number (or size) of all objects assigned to either memory. Another type of constraint can restrict the number of accesses to either memory in the most time-critical

parts of the program. Furthermore, it may improve performance if a few frequently used objects are replicated in both memories.

Unfortunately, compilers capable of exploiting the memory bandwidth increase offered by dual memory banks are not generally available. Most of the research for optimizing compilers has been oriented towards general-purpose processors with less instruction-level parallelism and a friendlier ensemble of registers.

The aim of this paper is to investigate the use of a *Scatter Search* for solving the dual memory bank assignment problem. Scatter Search is an evolutionary method that has been successfully applied to hard optimization problems [1]. In general, scatter search differs from other evolutionary procedures, such as genetic algorithms, by providing unifying principles for joining solutions based on generalized path constructions (in both Euclidean and neighborhood spaces) and by using strategic designs where other approaches resort to randomization. Additional advantages are given by intensification and diversification mechanisms that exploit adaptive memory.

An important idea is the way we approach memory assignment. We assume that all data objects to be assigned to memory are known. However, we use detailed frequency information, provided by the front end, to favor objects appearing in blocks in the critical path. Our approach is global, considering all data objects in all blocks simultaneously. Issues relating to commutativity, or the irregular register structure of the machine, are taken into account when assigning objects to particular memories. Finally, mechanisms exist for dealing with pre-assigned objects, and for balancing the load between memories.

The remainder of the paper is organized as follows. In Section 2, we explore the problem in greater depth. Examples throughout this section will be based on two commercial DSPs: The Motorola DSP56000 (M56K) and the highly irregular SGS-Thomson ST18950 (SG950). In Section 3 we describe the related work. We then present a formal description of the memory assignment problem in Section 4. Section 5 provides a detailed description of the scatter search implementation used to find solutions to the memory assignment model. Experimental results are given in Section 6. Finally, the conclusions obtained are given in Section 7.

```
double FIR(in double A[], in double B[], in int tap)
int i;
double sum=0;

for(i=0; i<tap; i++)
    sum += A[i]*B[i];
return sum;
```

```
CLR     A           X:(R0)+, X0    Y:(R4)+,Y0    [1]
REP     N-1                                      [2]
MAC     X0, Y0, A   X:(R0)+, X0    Y:(R4)+, Y0   [3]
MACR    X0, Y0, A                                [4]
```

Figure 1: Example of Software Pipelining.

2 A Closer Look at the Problem

The general memory assignment problem can be stated as the task of assigning data objects to one or both memories so that the performance of the final program is optimized and all constraints are satisfied. Though simple to state, a number of interrelated issues make finding a solution non-trivial. To understand these issues, the following series of brief examples are presented.

2.1 Software Pipelining

Figure 1 shows C-like code for implementing an N-Tap FIR filter as well as a portion of the resulting assembly-language code for the M56K. Notice that instructions 1 and 3 are *parallel* DSP instructions. For example, instruction 3 multiplies the contents of registers X0 and Y0 and adds the resulting product to the contents of accumulator A, retrieves new operands from memories X and Y into registers X0 and Y0, respectively, and updates address registers R0 and R4 using auto-increment for each. The assembly-language is characteristic of the software pipelining necessary for producing the fast, dense code required by DSP applications.

Our intent is to develop an algorithm that takes full advantage of the available parallelism exhibited by DSP architectures. However, there are a number of issues that make partitioning data into dual memory banks non-trivial. These issues are discussed in the subsections that follow.

2.2 Register Connectivity and Operator Commutativity

Non-commutative instructions require their operands to appear at specific ports of the ALU and are especially affected by the degree to which registers connect specific ports of functional units and memories. Register connectivity varies from machine to machine. Some architectures are relatively symmetric in that every operand and accumulator register can access both memories and both sides of the ALU. This flexibility greatly simplifies the task of memory assignment, since operands can be readily retrieved from either memory to appear at either port of the ALU. However, memory assignment is harder for more irregular architectures (like the ST950 and M56K).

For example, consider an architecture that has no direct path between both memories and both sides of the ALU. Now, consider the two non-commutative instructions, `A-B` and `C-A`, where variables `A`, `B`, and `C` reside in memory. If variable `A` is initially assigned to the right memory, it must first be loaded into one of the registers associated with the right memory, and then copied into one of the registers associated with the left side of the ALU. A similar problem exists if variable `A` is originally assigned to the left memory. In both cases, the only way to avoid an extra copy operation (and possibly an extra control step) is to duplicate A in both memories.

A comparable dilemma can also arise with commutative operations, if the machine has restricted register connectivity. For example, the expressions `A+B`, `B+C`, `A+C`, exhibit a "circular" dependency that makes it impossible to place single copies of `A`, `B`, and `C` in two memories so that operand pairs are always drawn from opposite memories. The rearrangement permitted by commutativity does not resolve the problem.

2.3 Data Duplication

Consider the high-level statement: `F[i] = F[i-1] + F[i-2]`. Notice how all three array references are to different elements of the same array. If the array is assigned to a single memory, each operand must be retrieved sequentially on separate steps - possibly increasing schedule length. Duplicating the array in both memories, however, allows both operands to be accessed in parallel on the same control step.

Unfortunately, duplication is expensive, as it requires twice as much memory to store both copies of the array. In practice, the limited memory capacity of most commercial DSPs limits the amount of data that can be duplicated. Furthermore, if the array is updated, as is the case here, an extra store operation is required to update both copies of the array. This may result in an increase in schedule length if the extra store cannot be performed in parallel with another ALU operation.

In general, duplication should only be performed when it leads to an overall improvement in performance.

2.4 Other Considerations

Constraints relating to the physical size of each memory may also exist. In order to keep performance high, designers of embedded systems, where DSPs are employed, favor the use of on-chip memory and avoid external memory as much as possible. This necessitates the judicious assignment of data if the capacity of each on-chip memory is not to be exceeded. Furthermore, certain program variables (usually arrays) may have to be assigned a priori to specific memories. In fact, some variables may have to be placed at specific locations that are accessed by other hardware components within the embedded system. At the lowest level, such decisions have nothing to do with the instruction set of the target machine.

For code blocks in critical paths, it is wise to balance the load between both memories. Each memory usually has a limited number of dedicated address and offset registers. Too many references to one memory may exhaust a memory's register resources and may even extend the schedule to accommodate one memory's activity. Moreover, high address register usage may require frequent re-initializations of the memory's address and offset registers. These "extra" instructions also lengthen the schedule and degrade performance. By finding a balance between both memories this undesirable behavior can be curtailed to some degree.

2.5 Summary

To summarize, program variables should be assigned to memory in order to support simultaneous accesses. However, the use of duplication should only be performed when the performance improvement justifies the additional memory cost. If the register structure of the target machine is irregular, operands of non-commutative operations should be assigned to mem-

ory in a way that minimizes the number of additional copy instructions required, because of data appearing in the "wrong" memory. Finally, the on-chip memory capacities of the DSP must be balanced and cannot be exceeded, and some variables may already be assigned to, or prohibited from, specific memories.

3 Related Work

Most approaches employ a *graph-oriented* model of the problem. For example, in [10] [11] Sudarsanum and Malik develop an approach based on simulated annealing that is sensitive to irregular register structures. The algorithm labels the nodes of a constraint graph where each node represents a symbolic register and memory preference, and each edge represents dependencies and constraints between the references. Although this approach is able to produce good results, high run times can be an issue due to the nature of simulated annealing.

A simpler version of the problem is solved by Saghir et. al. [8] [9]. The approach assumes that no restrictions apply to the registers that can be used, and operates only on basic blocks. An undirected interference graph is formed where nodes represent program variables and edges indicate pairs of variables that should be assigned to opposite memories. Once constructed, the interference graph is searched using a greedy algorithm to partition the nodes into two sets, one for each memory, having minimal dependence (fewest edges) among nodes within each set.

A limitation of using an interference graph is that it does not always provide a complete picture of the memory accesses that can occur in parallel during scheduling. This shortcoming is addressed in [12], where Zhuge et. al. employ a novel variable independence graph that is refined (by removing edges between memory accesses that cannot be scheduled in the same control step) to show potential parallel accesses that may occur later during scheduling.

In the work of Cho [2] et. al., traditional graph coloring and minimum-spanning tree algorithms with special constraints added to deal with issues arising from the non-orthogonal nature of the DSP are used to find a good memory assignment.

In [7] Sipkova presents a partitioning scheme based on the concepts of an interference graph, where partitioning of the interference graph is modeled as a Max Cut problem. The experimental results demonstrate that the partitioning algorithm finds a fairly good assignment of variables to memory banks. For small kernels from the DSPstone benchmark suite the performance is improved from 10% to 20%, for FFT filters by about 10%.

More recently, we [4] presented a novel, multi-objective model for the dual-memory assignment problem. In general, this work differs from the previous works in the following ways. First, a global optimization is performed for all blocks of code and all objects simultaneously. Detailed frequency information, provided by the front end, is used to favor objects appearing in blocks in critical paths. Issues relating to commutativity, or the irregular register structure of the machine, are taken into account when assigning objects to particular memories. In addition, mechanisms exist for dealing with preassigned objects, and for balancing the load between memories.

Unlike previous works, our model is not graph based. Rather, we model the problem of assigning objects to dual memories as a constrained optimization problem, and use a genetic algorithm to find a solution. The primary advantage of our model is that it allows different and, often conflicting, issues to be integrated and explored in a unified manner. The primary weakness of our approach is the genetic algorithm's ability to consistently find high-quality solutions in reasonable amounts of time.

In this paper, we extend our previous work and propose a novel scatter search procedure for solving our multi-objective model. The primary advantage of the scatter search over the GA is its ability to consistently find high-quality solutions with little computational effort. In addition, the flexibility that the scatter search provides with respect to how candidate solutions are encoded is exploited to reduce the number of constraints that are part of the model. Our results show that the scatter search is able to consistently outperform the GA both with respect to runtime and quality of solutions. Moreover, when tested with standard DSP kernels and applications available in the literature, the scatter search is able to achieve a 54% reduction in the number of memory cycles and a reduction in the range of 7% to 42% in the total number of CPU cycles.

4 The Model

4.1 Operational Assumptions

Like *all* of the other approaches described in the previous section, the scatter search implementation presented here models *static* situations and does not *directly* cope with arbitrary dynamic behavior, e.g., dy-

namic memory allocation or caching. It can, however, accommodate *statistically estimated* dynamic behavior. In our model, we use expected frequencies to summarize the net effect of statistically varying control flows. During a single execution, the actual access and update frequencies will differ from values in the model, but over several executions, average observed frequencies should approach those in the model.

More specifically, our model assumes that an analysis has been done by the front end to reveal the global histories (lifetimes and activities) of all data objects, both simple and complex. The information provided by the front end includes measures on how frequently a variable appears on the left (or right) side of a non-commutative (or unary) operator; the frequency with which a variable is updated; and, how frequently two variables appear together as operands of the same commutative operation. This frequency information is used to favor variables appearing in blocks on critical paths.

Determining the frequency information is made easier by the fact that most of the algorithms that use programmable chips have regular, well-understood behavior. Moreover, simulation, earlier experience with the algorithm, or allowing for the inclusion of branch probabilities and estimates of expected or maximum loop repetitions in the original source program may provide a precise performance profile.

4.2 Formulation

We model the dual memory assignment problem as an optimization problem with linear equality and inequality constraints. In our model, solution variables are index by i and j and are defined as follows:

Variable	Type	Meaning
x_i	0-1	if 1, variable i is assigned to the X memory
y_i	0-1	if 1, variable i is assigned to the Y memory
b_i	0-1	if 1, variable i is assigned to both X and Y memories
a_{ij}	0-1	if 1, both i and j are assigned to memory X only; 0 otherwise
b_{ij}	0-1	if 1, both i and j are assigned to memory Y only; 0 otherwise

The following are index sets of scalar objects:

Index Set	Meaning
N_i	number of "space units" (e.g., bytes, words, etc.) occupied by variable i. The units depend on the smallest data unit and most confined memory access mode of the machine.
L_i	total expected frequency of non-commutative (or unary) operators that access operand i on the left side
R_i	total expected frequency of non-commutative (or unary) operators that access operand i on the right side
U_i	total expected frequency of operations that update variable i
P_{ij}	total expected frequency of commutative binary operations that use operands i and j as paired operands
Max_x	capacity (in bytes) of X memory
Max_y	capacity (in bytes) of Y memory

L_i, R_i, U_i, and P_{ij} are constants derived directly from the usage of the variables in the original program and represent frequency-dependent numbers taken from the entire scope of the variable. Note that for operations that either consume or update a "variable" of size N(e.g., say an array) the effect or importance of the operation can be captured by multiplying the simple frequency by N_i when computing L_i, R_i, U_i, and P_{ij}.

4.3 Basic Constraints

In the constraints that follow, we refer to the "left" memory as the X memory and the "right" memory as the Y memory. Collectively, constraints 2 through 5 relate a_{ij} and b_{ij} to the other variables and ensure that a_{ij} and b_{ij} cannot both be equal to 1 at the same time. If $a_{ij} = 1$, both variables i and j are found in the same (X) memory and cannot be drawn from opposite memories − notably on the same control step. A similar observation applies to i and j with regards to the Y memory if $b_{ij} = 1$.

1. A variable i must be assigned either to the X memory, the Y memory or both memories:

$$\forall i: \quad x_i + y_i + b_i = 1$$

2. If variable i is assigned to the Y memory or both memories, it is never assigned *exclusively* to the X memory along with any other variable j:

$$\forall i, j: \quad 2(1 - a_{ij}) \geq y_i + b_i + y_j + b_j$$

3. If variable i is assigned to the X memory or both memories, it is not assigned exclusively to the Y memory:

$$\forall i, j: \quad 2(1 - b_{ij}) \geq x_i + b_i + x_j + b_j$$

4. If variable i and variable j are both assigned to the X memory, they must appear together only in the X memory, forcing $a_{ij} = 1$:

$$\forall i, j \; where \; i < j \; : \quad x_i + x_j \leq 1 + a_{ij}$$

5. If variable i and variable j are both assigned to the Y memory, they must appear together only in the Y memory, forcing $b_{ij} = 1$:

$$\forall i, j \; where \; i < j \; : \quad y_i + y_j \leq 1 + b_{ij}$$

6. The total size of the variables assigned to a memory cannot exceed the size of the on-chip data memories:

$$\sum_i N_i(x_i + b_i) \leqslant Max_x$$

$$\sum_i N_i(y_i + b_i) \leqslant Max_y$$

The objective function is as follows:

$$min\left[\sum_i y_i \cdot L_i + \sum_i x_i \cdot R_i + \sum_i b_i \cdot U_i + \sum_{i,j}(a_{ij} + b_{ij}) \cdot P_{ij}\right]$$

The value of the objective function reflects the cost of having to insert extra register copy instructions or to delay processing in order to move data to or from memory. More specifically, the objective function seeks to minimize a weighted sum that reflects the cost of having an operand appear in the "wrong" memory possibly requiring an extra control step due to a register copy (first two terms), the cost of having to update an operand twice if it appears in both memories possibly an extra control step due to the additional store (third term), and, finally, the cost associated with fetching paired operands on separate control steps whenever those operands are only assigned to the same memory (an extra control step).

The memory assignment problem can have additional requirements, which include: having certain variables pre-assigned to a specific memory; requiring certain variables to be either in the same memory or in opposite memories as certain others; and constraints on what variables can or cannot be duplicated in both memories. These types of requirements can be handled in a straight-forward manner by introducing additional constraints into the original model. (For the sake of space, these constraints have been omitted.)

5 Scatter Search Implementation

Scatter search, from the perspective of meta-heuristic classification, may be viewed as an evolutionary (or population-based) algorithm. However, there are a number of important features that distinguish scatter search from other well-known evolutionary algorithms, like Genetic Algorithms (GAs). For example, unlike genetic algorithms which are based on the analogy of Darwinian evolution, scatter search derives its foundations from strategies originally proposed for combining decision rules and constraints (in the context of integer-linear programming). Scatter search is based on the belief that (i) useful information about the form (or location) of optimal solutions is typically contained in a suitably diverse collection of elite (high-quality) solutions; and (ii) that the information in these elite solutions can best be exploited by combining multiple solutions simultaneously to extrapolate beyond the regions spanned by the solutions considered. Both convex and non-convex solution combinations are allowed, thus simultaneously facilitating the exploitation and exploration of the problem's search space.

More specifically, scatter search operates on a set of solutions, called the *reference set*, by combining these solutions to create new ones. The most common mechanism for combining solutions is one where a new solution is created from a linear combination of one or more existing solutions. New solutions can gain admittance to the reference set based on their quality or diversity. Unlike the population in a genetic algorithm, the reference set of solutions in scatter search tends to be small. A typical population size in a genetic algorithm consists of 100 solu-

tions, while the reference set in in scatter search often has 20 (or fewer) solutions. Moreover, genetic algorithms typically produce new solutions by randomly selecting (sampling) two solutions from the current population then applying a recombination operator (e.g., "crossover") to generate one or two offspring. In contrast, scatter search chooses *two or more* solutions from the reference set in a *systematic* (i.e., non-random) way with the purpose of creating new solutions. Since the combination process considers at least all pairs of solutions in the reference set, there is a practical need to keep the reference set small.

Our scatter search procedure for solving the memory assignment problem is based on the scatter search template described by Laguna in [1], and combines the following five methods:

- A *Diversification Generation* method to generate a collection of diverse trial solutions, using an arbitrary trial solution (or seed solution) as an input.
- An *Improvement* method to transform a trial solution into one ore more enhanced trial solutions. (For the memory-assignment problem, the input solution is not required to be feasible, but the output solution is expected to be feasible.)
- A *Reference Set Update* method to build and maintain a *reference set* consisting of the b "best" solutions found (where the value of b is typically small, e.g., no larger than 20), organized to provide efficient accessing by other parts of the method. Solutions gain membership to the reference set according to their quality and diversity.
- A *Subset Generation* method to operate on the reference set, to produce a subset of its solutions as a basis for creating combined solutions.
- A *Solution Combination* method to transform a given subset of solutions produced by the Subset Generation Method into one ore more combined solutions.

The basic scheme, and the interaction between the methods, is illustrated in Fig. 2. The first stage of the scatter search procedure begins by generating a diverse set of high-quality solutions. This is accomplished using the Diversification Generation and Improvement methods. The goal of the diversification method is to produce solutions that differ from each other in significant ways, and that yield productive alternatives in the context of the problem considered. This can be viewed as sampling the solution space in a systematic fashion to identify high-quality solutions with controllable degrees of difference. The goal of the improvement method is to improve the quality of the initial solutions, by using any one of a variety of optimization-based or heuristic-based improvement approaches that have been found effective for the problem at hand. The initial set of solutions is created by iteratively generating new solutions by invoking the Diversification method followed by the Improvement Method. The number of initial solutions generated is typically ten times the size of the actual number that will eventually enter the reference set.

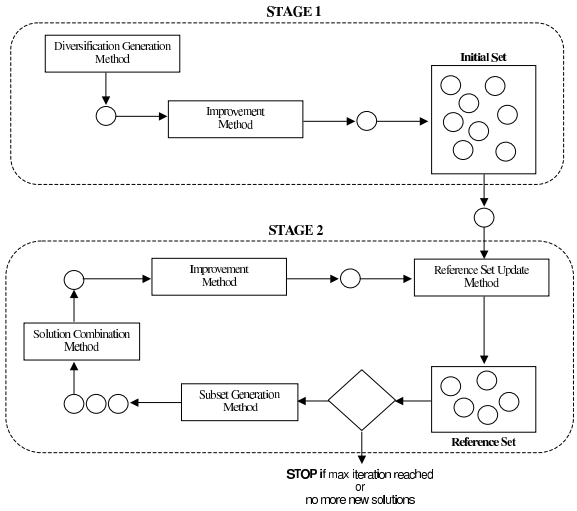

Figure 2: Typical Scatter Search Algorithm.

With the initial set of solutions in place, the second stage of the procedure begins with an application of the Reference Set Update method. This method typically accompanies each application of the Improvement method and is used, on the first iteration, to designate a subset of the initial solutions to the reference set. The primary purpose of creating and maintaining a reference set is to have a current collection of diverse, high-quality solutions at hand for generating further solutions through a combination strategy.

The Subset Generation method is then used to systematically group the solutions in the reference set into subsets of two or more individuals. Individuals are chosen to produce points both inside and outside the convex regions spanned by the reference solutions. Once determined, the Solution Combination method is used to transform each subset of solutions produced by the Subset Generation method into one (or more) combined solutions. As there is no guarantee that

the new solutions are high-quality, the Improvement method is used to improve the quality of the new solutions. The Reference Set Update method is then applied once more. At this step, a new improved solution can replace an existing solution in the reference set, but only if it has superior quality or diversity.

The entire process in the second stage repeats until a maximum number of iterations has been performed, or until the reference set does not change. The "best" solution in the reference set is reported as the final solution.

In the subsections that follow, we describe the implementation of each of scatter search methods as they apply to the memory-assignment problem. However, we begin with a brief description of how solutions are represented (i.e., encoded).

5.1 Solution Representation

We employ two different representations depending upon whether or not the designer allows variables to be duplicated in both memories. In the case where duplication is *not* allowed, each solution $S = (S_1, S_2, S_3, ..., S_n)$ is represented as a string of length n, where n is the number of variables to be assigned to one or possibly both memories. The string stores the memory assignment in binary form. If $S_i = 0$, variable i is assigned to the X memory (i.e., $x_i = 1$). If $S_i = 1$, variable i is assigned to the Y memory (i.e., $y_i = 1$). Notice that, if $S_i = 0$ and $S_j = 0$, both data objects i and j are assigned only to the X memory (i.e., $a_{ij} = 1$). Similarly, if $S_i = 1$ and $S_j = 1$, data objects i and j are assigned exclusively to the Y memory (i.e., $b_{ij} = 1$).

When duplication of variables *is* allowed, we represent the memory assignment for variable i as a tuple *(x,y)*, where *(0,1)* means variable i is assigned to the y memory, $(1, 0)$ means that variable i is assigned to the x memory, and $(1, 1)$ means that variable i is duplicated in both memories. The memory assignment for all n variables is represented as a string of tuples of length n.

The advantage of the previous representations is that they avoid having to deal directly with constraints 1-5 described in Section 4.3. The only constraint that remains is constraint 6 which ensures that the capacity of each memory is not exceeded.

5.2 Diversification Method

We employ a systematic (i.e., non-random) procedure for generating the initial pool of trial solutions that is similar to that first proposed in [1] for the 0-1 knapsack problem. We let x denote an n-vector, each of whose components x_i receives the value 0 or 1. Based on this definition, we employ a diversification method that takes x as its seed solution, and generates a collection of solutions associated with an integer $h = 2, 3, ...h_{max}$, where $h_{max} \leqslant n - 1$. Although it is recommended for h_{max} to be less than or equal to $n/5$, when a large number of diverse solutions is needed, the value of h_{max} must approach n-1.

We generate two types of solutions, x' and x'', for each value of h, by the following rule:

- **Type 1 Solution**: Let $x'_{1+kh} = 1$ - x_{1+kh} for $k =$ 0,1,2,3,...,$\lfloor n/h \rfloor$, where $\lfloor k \rfloor$ is the largest integer satisfying $k \leq n/h$. All other components of x' are equal to x (i.e., the seed solution).
- **Type 2 Solution**: Let x'' be the complement of x'

h	x'	x"
seed	(0,0,0,0,0,0,0,0,0,0)	(1,1,1,1,1,1,1,1,1,1)
2	(1,0,1,0,1,0,1,0,1,0)	(0,1,0,1,0,1,0,1,0,1)
3	(1,0,0,1,0,0,1,0,0,1)	(0,1,1,0,1,1,0,1,1,0)
4	(1,0,0,0,1,0,0,0,1,0)	(0,1,1,1,0,1,1,1,0,1)
5	(1,0,0,0,0,1,0,0,0,0)	(0,1,1,1,1,0,1,1,1,1)

Table 1: Diverse pool of binary strings.

To illustrate, Table 1 shows the initial pool of trial solutions that would be generated using the seed x = (0,0,..,0) and the values h = 2,3,4,5. The first row corresponds to the seed and its complement and the remaining rows to the corresponding h values. The progression in Table 1 suggests the reason for preferring $h_{max} \leq n/5$. As h becomes larger, the solutions x' for two consecutive values of h differ from each other proportionally less than when h is smaller. An option to avoid diminishing diversity is to allow h to grow by increasing steps for larger values of h.

5.3 Improvement Method

Since the solutions generated are not guaranteed to be feasible, the improvement method should be capable of handling starting solutions that are either feasible or infeasible. We employ a local search consisting of two deterministic phases:

- ***Phase 1:*** If the trial solution is infeasible, the local search attempts to repair it by first finding a feasible solution. In particular, if the capacity of either the X or Y memory is exceed, variables are selected to be moved to the opposite memory until the capacity constraint of either memory is no longer violated. All variables in each memory are sorted in the decreasing order of their memory preference to size ratios; i.e., R_i/N_i for the X memory and L_i/N_i for the Y memory. The procedure always chooses the last variable from the list of available variables that will fit in the opposite memory and moves the variable to the opposite memory. Once the solution becomes feasible, the second phase is applied.

- ***Phase 2:*** The second phase of the local-search strategy aims to improve the quality of the feasible solution by employing a *greedy* algorithm to swap variables between memories. The algorithm is greedy in the sense that on each iteration, the variable that can potentially improve the cost of the objective function by the largest amount is considered for reassignment into the other (target) memory. When a variable is reassigned into the opposite memory, one (or more) variables from the target memory are also swapped back into the memory from which the original variable was taken. This ensures that the capacity of each memory is not exceeded. It also tends to balance the amount of data stored in both memories.

5.4 Reference Set Update Method

The purpose of the Reference Set Update method is to create and maintain a set of reference solutions. In our implementation, we partition the reference set ($RefSet$) into two subsets: $RefSet_1$ and $RefSet_2$. $RefSet_1$ contains b_1 high-quality solutons, while $RefSet_2$ contains b_2 diverse solutions, with $|RefSet| = b_1 + b_2$. According to these definitions, the construction of the original reference set starts with the selection of the highest-quality b_1 solutions from the original set of solutions. (Recall that the initial set of solutions are constructed with the application of the Diversification Generation method followed by the Improvement method during the first stage of the search.) The quality of a solution is determined by its objective function value (see Section 4.3). As we are solving a minimization problem, the lower a solution's objective value, the better a memory assignment it represents. Thus, $RefSet_1$ contains the b_1 solutions with lowest objective function values in the initial set of solutions.

In order, to find diverse solutions to add to $RefSet_2$, it is necessary to define a diversity measure. In general, let $d(x, y)$ be the distance of two solutions (i.e., memory assignments). The degree of diversity is defined in terms of the minimum distance $d_{min}(x) = min\{d(x, y)|\forall y \in RefSet_1 \cup RefSet_2\}$. Thus, each solution in $RefSet_2$ has the largest minimum distance from any other solution in the reference set.

To calculate the distance between two solutions, we consider the distance between two solutions as the sum of the absolute difference between its corresponding values; i.e., $d(x, y) = \sum_{i=1..n} abs(x_i - y_i)$. For example, the distance between the two solutions (0,1,1,1,0,0,0,0,1) and (0,1,0,0,1,0,0,1,0,0) is calculated as follows:

$$\begin{array}{l} (0,1,1,1,0,0,0,0,0,1) \\ (0,1,0,0,1,0,0,1,0,0) \\ \hline (0,0,1,1,1,0,0,1,0,1) = 5 \end{array}$$

We then use this distance measure to select b_2 solutions to complete the initial reference set. In particular, we look for a solution that is not currently in $RestSet_1 \cup RefSet2$ and that maximizes the minimum distance to all the solutions currently in the reference set.

During the second stage of the scatter search algorithm, the new improved solutions produced by the Solution Combination and Improvement Methods seek to gain entry to the reference set. When performing a reference set update, new solutions are first considered for membership in $RefSet_1$. If their quality (objective function value) precludes their entry, they are then considered for entry into $RefSet_2$ based upon their diversity, as determined above. If a new improved solution is admitted to the reference set, it always replaces the "worst" solution in the reference set, thus keeping the size of the reference set constant.

5.5 Subset Generation Method

The Subset Generation method operates on the reference set to produce a subset of its solutions as a basis for creating combined solutions. Given that the reference set contains both high-quality and highly-diverse solutions, scatter search generates the following subsets: (i) all 2-element subsets; (ii) 3-element subsets derived from 2-element subsets by augmenting each 2-element subset to include the best (lowest objective value) solution not in this subset; (iii) 4-element

subsets derived from 3-element subsets to include the best (lowest objective value) solution not in this subset; (iv) subsets consisting of the best i elements for $i = 5$ to $(b_1 + b_2)$. The memory assignments in each subset are combined to produce a new memory assignment, as described next.

5.6 Solution Combination Method

The Solution Combination method uses the subsets generated with the Subset Generation method to combine the elements in each subset with the purpose of creating new trial solutions. We use a probabilistic rule to generate new trial solutions. Specifically, our combination method calculates a score for each variable (S_i) in the string, based on the objective function value of the two (or more) reference solutions being combined. The score for variable i that corresponds to the combination of the reference solutions j and k is calculated as follows:

$$score(i) = \frac{ObjVal(j)x_i^j + ObjVal(k)x_i^k}{ObjVal(j) + ObjVal(k)}$$

Where $ObjVal(j)$ is the objective function value of solution j and x_i^j is the value of the i^{th} variable in solution j. Then, the trial solution is constructed by using the score as the probability for setting each variable to one, i.e., $P(S_i = 1) = score(i)$. This can be implemented as follows:

$$x_i = \begin{cases} 0 & \text{if } r \leq score(i); \\ 1 & \text{if } r > score(i); \end{cases} \quad (1)$$

where r is a uniform random number such that $0 \leq r \leq 1$. Note that the combination mechanism described here may construct infeasible solutions. However, this does not represent a problem, as the Improvement method is always applied to each trial solution created after the application of the Combination method. Recall that the Improvement method is designed to deal with either feasible or infeasible solutions.

6 Experimental Methodology

Our experimental methodology consists of two phases. During the first phase, the performance of scatter search is compared with that of the genetic algorithm described in [4]. The second phase involves testing the performance of scatter search using various kernels from the DSPstone benchmark suite [14], along with some applications.

6.1 Comparison of scatter search and GA

In this subsection, we compare the performance of scatter search presented here to the genetic algorithm in [4]. To allow for a fair comparison, both algorithms were implemented in C++, compiled under Cygwin with the gcc compiler version 2.96, and executed on the same Windows 2000 system with 2.4-GHz Intel Pentium 4 processor with 512 Megabytes of RAM.

In the experiments that follow, *run time* is expressed in seconds and *cost* is the objective function (see Section 4.3) value of the best solution found (where a lower objective-function cost is considered better). In all experiments, the GA was executed using the same operational parameters reported in [4]. In particular, a population size of 50, and mutation and crossover rates of 0.05 and 0.75, respectively, were used. The scatter search algorithm, on the other hand, always used a reference set of size 12 with $RefSet_1$ containing 10 high-quality solutions and $RefSet_2$ with 2 diverse solutions. (The previous parameter values were found to be the most effective after a significant amount of testing.) In each run, scatter search and the genetic algorithm were allowed to run for a maximum of 1000 generations or until the reference set or population, respectively, converged.

Each algorithm was tested by running the algorithm fifteen times on each of the 48 problem instances first reported in [4]. To simplify the analysis, the various problem instances are divided into 3 classes, denoted by A, B, and C, according to their sizes. All problems are subject only to the constraints in Section 4.3. In class A, each problem has 50 data items. The number of items where the left memory is preferred is either 25% or 50%. Similarly, the number of items where the right memory is preferred, and the number of items that require updating is either 25% or 50%. Finally, the number of commutative binary operations that use operands i and j as paired operands is either 50% or 75%. As all combinations exist, class A contains 16 problem instances. Classes B and C are similar to class A, but contain more data items. Class B contains 100 items, while class C contains 200 items.

The results for all three classes are given in Table 2. We utilized a t-test [13] to determine whether

the difference between the average genetic and scatter search results are significant. Table 2 shows those problem instances in classes A, B, and C that were determined to be insignificant at a 95% confidence level (denoted by EQUAL). For class A, three of the problem instances (i.e., 2, 7, and 8) were found to yield similar results. As the problem size increases from class B to C, the difference between the means also increases, and for class C only one of the means is statistically similar. Thus, for larger problem instances the expected outcome of the scatter search is clearly better than the genetic algorithm approach. Moreover, since only 10 of the 48 problems were found to be similar, it can be concluded that the scatter-search results are better over all problem instances.

We also calculated the confidence intervals about each of the genetic and scatter search means (at a 95% confidence level). The confidence intervals for the genetic algorithm results are much wider than those for the corresponding scatter search. Therefore, the scatter search results are more reliable since there is a narrower range around the reported results compared to those of the genetic algorithm.

Finally, in every case, the average run time of the scatter search is less than the time of the GA. Overall, our results show that the scatter search is capable of finding high-quality solutions for all problem instances considered in a moderate amount of time. This is not the case for the genetic algorithm. As the problem size increases through the Classes A, B, and C the above observations become increasingly pronounced.

6.2 Results Found Using Standard Benchmarks

We now return to the original memory-assignment problem and test the scatter search using various kernels from the DSPstone benchmark suite [14], along with some applications. In the embedded systems community, DSPs are typically evaluated using loops that form the core of many common signal-processing algorithms. Each kernel contains one or more loops with operations on two or more global arrays. For each kernel (and application), three variants were considered. In the first version, data objects were assigned explicitly only to one (i.e., the X) memory bank. In the second version, data objects were partitioned (using scatter search) between the X and Y memories. Finally, in the third version all data objects were duplicated in both (X and Y) memories.

Based on the memory assignment information, code was produced for the Motorola DSP56000. Table 3 shows the performance results obtained for the selected DSPstone kernels (and applications). For each version, the first column shows the total number of clock cycles executed (T_C); the second and third columns show the number of clock cycles resulting from accesses to the X (X_C) and Y (Y_C) memory banks, respectively. (Note: The total number of clock cycles required for each program and memory assignment were computed using the instruction timings provided in [5]).

It can be observed that the best overall performance gain is achieved when the scatter search is used to partition the data into different memories. The number of memory access cycles for both the X and Y memory are reduced by approximately 54%, while the reduction in the total number of clock cycles ranges from 7% to 42%. These results are optimal [5], and are a direct consequence of the fact that most of the kernels (and applications) provide ample opportunities to exploit instruction-level parallelism through judicious memory assignments. Nonetheless, Table 3 indicates that for some programs (e.g., fib2) the best software solution for providing parallel accesses is to duplicate the data in both memories. However, as discussed in Section 2.4, duplication of data may result in an increase in schedule length if the extra store operations required to update both copies of the data cannot be performed in parallel with another ALU operation. Therefore, it is clear from Table 3 that there is no benefit in duplicating all data indiscriminately when comparable or superior performance could be obtained with far less memory by judiciously allocating arrays to the appropriate memory banks. In general, any gain in performance obtained by duplication must be weighed against the increase in memory cost.

7 Conclusions

In this paper, we have presented a novel methodology, based on scatter search, for assigning data objects to dual-bank memories. Our approach is global, and special effort is made to identify those objects that could potentially benefit from an assignment to a particular memory, or perhaps both memories. Issues relating to commutativity, or the irregular register structure of the machine, are taken into account when assigning objects to particular memories. In addition, mechanisms exist for dealing with preassigned objects, and for balancing the load between memories. The primary advantages of the scatter search

Table 2: Results for Classes A, B, and C

	Comparison	Genetic Algorithm			Scatter Search		
Prob	scatter search=GA?	Avg. Cost	+/- conf. Interval	Avg. Time (sec)	Avg. Cost	+/- conf. Interval	Avg. Time (sec)
A-1		22.4	3.5	11	19	0	0.8
A-2	EQUAL	39.8	1.3	10.8	39.6	0.4	1.6
A-3		67	7.25	11.6	64.2	0.39	1.2
A-4		91.6	2.7	10.4	90	0	0.8
A-5		36.8	4.7	11	34	0.32	1.2
A-6		53	1.1	10.6	52.6	0.25	1
A-7	EQUAL	45.2	0.7	11	45.2	0.20	0.8
A-8	EQUAL	55	0	10.4	55	0	1.2
A-9		56.6	6.7	10.4	54.6	0.7	1.2
A-10		73.2	4.6	12	70	0	0.8
A-11		80.2	8.2	11.8	74.2	0.2	1
A-12		65	4.6	11.2	63	1.0	1.2
A-13		39.2	6.1	11.6	32.4	0.25	1
A-14		40.6	4.8	11.6	38	0	0.8
A-15		55.4	3.9	10.6	54	0	1.2
A-16		91.2	16.1	11.8	78	0	1
B-1	EQUAL	62	7.8	29.6	61.2	0.8	2
B-2		99.4	12.8	30.4	87.6	0.2	6.2
B-3	EQUAL	170.6	5.2	30.8	171.2	2.0	2.8
B-4		173.6	7.7	32.4	167	0	1.8
B-5		86.8	11.3	28	79.6	0.9	3
B-6		114	47.4	28	68	0.5	3.6
B-7		100	7.6	30.2	93	0	2
B-8	EQUAL	201.6	31.7	29.8	195.6	2.1	2.8
B-9		101.5	32.0	30.2	88.6	2.4	5.8
B-10		185.2	28.2	30.4	174.8	2.7	5.8
B-11	EQUAL	170	26.5	30.4	163.6	1.7	3.4
B-12	EQUAL	267.2	43.4	30.2	267.4	1.7	3.8
B-13		122.8	15.5	30	117	2.0	5.4
B-14		128.8	1.3	30.4	124	0	3.2
B-15		238.0	32.0	30.6	202.8	8.7	5
B-16	EQUAL	266.6	11.9	28.8	267	1.3	4.2
C-1		449.8	36.3	60	461.6	2.5	32.8
C-2		510.6	35.5	59.4	490	0.8	20.2
C-3		448.4	20.9	61.8	424.8	5.7	20.8
C-4		800.8	37.1	61.2	780.6	6.8	16.6
C-5		383	33.0	60.6	347	5.5	17.6
C-6		505.8	102.2	59.6	457.2.2	8.3	31.2
C-7	EQUAL	378.8	54.9	59.2	370.6	4.8	24.8
C-8		894.6	86.5	61.4	803.2	3.4	24.8
C-9		581	94.9	60	405.4	5.6	19.2
C-10		756.6	59.6	60	730.2	9.9	41.4
C-11		603.2	114.9	59.2	528.2	2.4	25.6
C-12		1141.4	75.7	61.4	1068	8.1	31.8
C-13		412.8	141.6	61.6	298.4	10.1	39.4
C-14		737.4	66.5	60.4	665.6	7.7	31.4
C-15		740.4	30.4	63.4	631	2.2	25.8
C-16		777.6	57.2	60.6	778.4	15.6	33.6

include its speed, ability to produce high-quality solutions, and the ease with which different, and often conflicting, issues can be integrated and explored all within a unified model. Our experimental results show that our method is able to find a good memory assignment. When applied to typical DSP kernels and some applications, we were able to reduce the number of memory cycles by approximately 54% and the total number of cycles by 7% to 42%.

With regards to scatter search, the performance of the algorithm has been compared with that of the genetic algorithm reported in [4]. The experimental results show that the performance of the scatter search is superior to that of the GA both with respect to run time and quality of solutions, across all problem instances considered.

References

[1] M. Laguna and R. Marti, "Scatter Search: Methodology and Implementations in C," Kluwer Academic Publishers, ISBN 1-4020-7376-3, 2003.

[2] J. Cho, Y. Paek, and W. Whalley, "Efficient Register and Memory Assignment for Non-orthogonal Architectures via Graph Coloring and MST Algorithms", Proc. of the International Conference on LCTES and SCOPES, Berlin, Germany, 2002

	One Memory			Partitioning			Duplication		
Kernel/Applc'n	T_C	X_C	Y_C	T_C	X_C	Y_C	T_C	X_C	Y_C
fir2dim	4226	1922	0	2426	1011	911	2426	1011	1011
dotproduct	618	201	0	416	101	100	416	101	101
convolution	416	203	0	212	102	101	212	102	102
lms	516	172	0	388	67	105	388	99	106
biquad n sections	516	172	0	388	67	105	440	99	97
n complex updates	2820	800	0	2420	400	400	2420	600	400
mat10x10	6916	2100	0	4516	1100	100	4516	1100	100
mat1x3	56	21	0	38	9	12	38	12	12
fft1024	123060	60407	0	78154	30715	29692	121160	40955	39932
n real updates	228	64	0	212	32	32	212	48	32
app1	80	30	0	60	20	10	60	20	20
fib1	160	70	0	100	40	30	120	50	50
fib2	32	12	0	28	6	6	24	8	8

Table 3: Results for DSPstone Kernels and Applications.

[3] M. Garey and D. Johnson, "Computers and Intractability", A Guide to the Theory of NP-Completeness, New York: W.H. Freeman and Company, 1979.

[4] G. Grewal, S. Coros, A. Morton, D. Banerji, "An Evolutionary Penalty-Function Approach to Memory Assignment in the DSP Domain," 10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures, Austin, Texas, February 12, 2006 pp., 1-12.

[5] Benchmark Programs, DSP56000/DSP56001 Digital Signal Processor User's Manual, Motorola INC., Semiconductor Products Sector, DSP Division, Austin Texas.

[6] D. Orvosh, L. Davis, "Shall We Repair? Genetic Algorithms, Combinatorial Optimization, and Feasibility Constraints," in Proceedings of the Fifth Internatinal Conference on Genetic Algorithms, Morgan Kaufmann, San Mateo, CA, pp. 650, 1993.

[7] V. Sipkova, "Efficient Variable Allocation to Dual Memory Banks for DSPs," Springer Lecture Notes in Computer Science, Volume 2826, 2003.

[8] M. Saghir, P. Chow, and C. Lee, "Exploiting dual data-memory banks in digital signal processors," Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 30(5), pp., 234-243, 1996.

[9] M. Saghir, P. Chow, and C. Lee, "Automatic Data Partitioning for HLL DSP Compilers," Proc. of the 7th International Conference on Signal Processing Applications and Technology, pp., I-658-664, 1995.

[10] A. Sudarsanum and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs," International Conference on Computer-Aided Design, pp., 388-392, 1997.

[11] A. Sudarsanam and S. Malik, "Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs," Journal of ACM Transactions on Automation of Electronic Systems (TOADES), 5, pp., 242-264, 2000.

[12] Q. Zhuge, B. Xiao, and E. Sha, "Variable Partitioning and Scheduling of Multiple Memory Architectures for DSP," Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2002.

[13] I. Miller and M. Miller, "John E. Freud's Mathematical Statistics," 6th Edition, Prentice Hall, 1999.

[14] V. Zivojnovic, J. Velarde, C. Schaeger, and H. Meyr "DSPstone - A DSP oriented Benchmarking Methodology," In proceedings of the 6th International Conference on Signal Processing Applications and Technology, 1994.

[15] M. Mitchell, "An Introduction to Genetic Algorithms," Cambridge, MA, MIT Press, 1996.

Use of an Embedded Configurable Memory for Stream Image Processing

Michael G. Benjamin and David Kaeli
Northeastern University Computer Architecture Research Group
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA 02115
{mbenjami, kaeli}@ece.neu.edu

***Abstract*— We examine the use of the embedded Blackfin BF561 processor for high-definition image processing using the stream model of computing. The Blackfin features a configurable memory hierarchy that minimizes the Memory Wall effect. We describe the stream model and its application to the BF561 to utilize low-latency on-chip memory and compare to a worst-cast baseline using SDRAM only. We find a 2X to 3X speedup in the edge detection of a 1920x1080 pixel image using C and 3X to 11X speedup using assembly.**

I. Introduction

As high definition (HD) cameras and displays become commonplace, the need to quickly process large images increases. The embedded processors found in such devices have, for the most part, not followed the trend of increasingly faster clock speeds seen in general-purpose processors. High clock speeds translate to higher power usage, which in turn requires cooling systems to maintain reliable performance. Such requirements are ill-suited to the constrained environments of embedded systems. For this reason, and because embedded processors tend to be geared more to specific classes of applications, embedded chips have specialized hardware resources to do more per cycle, rather than reducing the cycle time.

Many image processing routines can be expressed as a set of instructions, called compute kernels, which transform raw image data into meaningful information. Vector processing units, special-purpose hardware such as video ALUs (vALUs), and convergent cores have extended some embedded architectures to allow multiple pixels to be transformed at once. But because of the vast amounts of data needing to be processed, memory performance is critical. Researchers have examined the memory hierarchy and proposed methods to minimize the latency associated with accessing high-capacity, slow off-chip SDRAM (called the Memory Wall effect [1], [2]). This need to feed the computational units which may otherwise be starved for data motivates our work and our examination of the stream model of computing.

In this paper, we examine two approaches to using low-latency L2 and L1 SRAM available in the memory hierarchy of the Analog Devices Inc (ADI) Blackfin BF561 embedded processor. The first uses SRAM to accelerate each kernel individually, and the second applies the stream model which uses SRAM for inter-kernel dataflow. We compare these approaches to a baseline of only using SDRAM for edge-detection in a HD image.

Section II discusses generic image processing algorithms and memory considerations. Section III explains the stream paradigm further. Section IV describes the ADI Blackfin embedded processor. Section V describes the two approaches for low-latency memory usage. Section VI describes the workload examined here. Section VII describes the implementation of an edge-detector using the BF561 and discusses results. Section VIII discusses the related work for utilizing configurable memory hierarchies and applying the stream model to existing architectures. Section IX concludes the paper. Section X outlines future work.

II. Image Processing and Memory

Many image processing routines can be summarized by the algorithm below, where mem_i and mem_o are memory locations; $rows$ and $cols$ are the number of rows and columns in the image, respectively:

```
ImageProcKernel (mem_i, rows, cols, mem_o)
1  for y ← 0 to rows − 1
2   do for x ← 0 to cols − 1
3     do p = GetPixel(x, y, mem_i)
4     Transform (p)
5     SetPixel(p, x, y, mem_o)
```

A program whose control flow consists of a sequence of such transformation kernels can itself be described with the following algorithm:

```
ProcessImage
1  foreach kernel_i in Control Flow
2    Run(kernel_i(origImage, rows, cols, procImage))
```

The Run routine simply runs the $kernel_i$ algorithm and transforms every pixel in *origImage* into the appropriate pixel in *procImage*. Each kernel's Transform routine can be optimized using SIMD instructions that make use of replicated hardware and media extensions to the architecture. But the GetPixel and SetPixel routines access memory structures

which often are located in off-chip SDRAM (especially for HD images which are much larger than many on-chip memories).

Better overall performance can be achieved by having needed data in low-latency, on-chip memory. Conventionally, caching has been used to move data into SRAM, but the usual cache benefits are limited for image processing. Images have two dimensional spatial locality, but caches only capture 1D locality [3], [4]. Also, due to its nature, image processing has limited temporal locality - this lack of data reuse can cause cache pollution.

Mechanisms at both the architecture and application have been developed to deal with this issue. Architecturally, split caches, stream buffers, and adaptive caches [5], [6], [7], [8] have been introduced to utilize both spatial and temporal locality. Scratch-pad memories have been presented as a cache alternative [9]. From the application perspective, frameworks and methodologies have been developed to increase data locality [10] and to perform source-to-source transformations to ensure [11] data is consumed soon after it is produced. Novel architectures have also been introduced to offload processing to the memory elements themselves such as vectorized intelligent RAM, processor-in-memory, etc. [12], [13].

An alternative approach has been described in the context of stream processors. Memory access to SDRAM only happens initially to get data, when it is compulsory to do so, and after the completion of all transformation kernels, with inter-kernel communication via on-chip local stores. The next section describes the stream model in detail.

III. Stream Computing Paradigm

The stream model of computing seeks to maximize locality while exposing parallelism [14]. Stream computing decouples memory and computation into streams and kernels, respectively. Streams consist of a set of data records which are to be processed. Kernels describe a sequence of instructions that are applied to each input record and whose results are saved into output records. A stream program is expressed as data streams which pass through a sequence of computational kernels.

Figure 1 shows an example of the dataflow of a generalized stream program. For example, many image processing programs can be described using a stream of pixel blocks and a set of n transformation kernels.

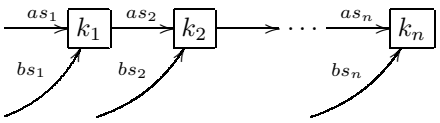

Fig. 1. An example dataflow graph for a generic stream program.

Locality is maximized during a kernel's execution, because all data accesses can be served by a local memory store (kernel locality) and because results produced by one kernel are quickly consumed by the next (producer-consumer locality). Given the computational resources, multiple kernels can operate simultaneously in a pipeline, exposing thread-level parallelism. For each thread and within a kernel, independent instructions can operate at the same time, exposing instruction-level parallelism. A particular instruction could be vectorized and applied to many elements of the data stream using SIMD hardware, exposing data-level parallelism [15].

The rising demand for media processing has motivated the development of a number of architectures and re-examination of existing architectures to work with the stream model. Architectures such as Imagine [16], [17], Merrimac [18], and Stream Processors Inc.'s Storm-1 [19] have been developed explicitly to exploit the stream model.

Also, to support diverse, dynamic applications, architectures that can be reconfigured to execute efficiently for many classes of applications have been introduced. Architectures such as TRIPS [20] and RAW [21] can be described as polymorphous - that is, they can morph between a number of operating modes, each capturing some class of applications. These polymorphous architectures can be programmed to achieve the stream model's high locality and parallelism.

To program all of these architectures, a number of streaming languages have been developed, including Stream-C/Kernel-C for Imagine, StreamIT [22] for RAW, and the Stream Virtual Machine specification [23], which uses two-level compilation for code written using a supported stream language and a number of architectures (including TRIPS, RAW, Smart Memories, Imagine, and even graphics processing units). There are also performance studies comparing streaming and intelligent memory architectures [24], [25] The stream model has also been more formally studied with respect to other computing models like Kahn process networks and dataflow [26].

IV. Blackfin Processor

The Blackfin is an embedded media processor based on the Micro Signal Architecture [27] developed jointly by Analog Devices (ADI) and Intel. The Blackfin is a fixed-point, convergent architecture that provides both micro-controller (MCU) and digital signal processing (DSP) functionality in a single processor. The MCU functionality is provided with a 32-bit variable-length RISC instruction set supporting vector operations (add/subtract, multiply, shift, etc.) and vector video operations (add/subtract, average, sum-of-absolute-differences, etc.). The DSP functionality is provided via two multiply-and-accumulate (MAC) units accessible via an orthogonal instruction set allowing for up to three instructions to be issued in parallel.

Figure 2 shows the basic units of the Blackfin core: the address arithmetic unit (with appropriate data address and pointer registers, and data address generators), the control unit, and the data arithmetic unit (with data register file, MAC units, vALUs, and ALUs). Also, given the constrained environment in which embedded systems exist, the Blackfin includes software-programmable on-chip PLL, dynamic power management to vary frequency and voltage, multiple operating modes, and memory management unit. The Blackfin hardware supports 8-bit, 16-bit, and 32-bit arithmetic operations - but is optimized for 16-bit operations [28].

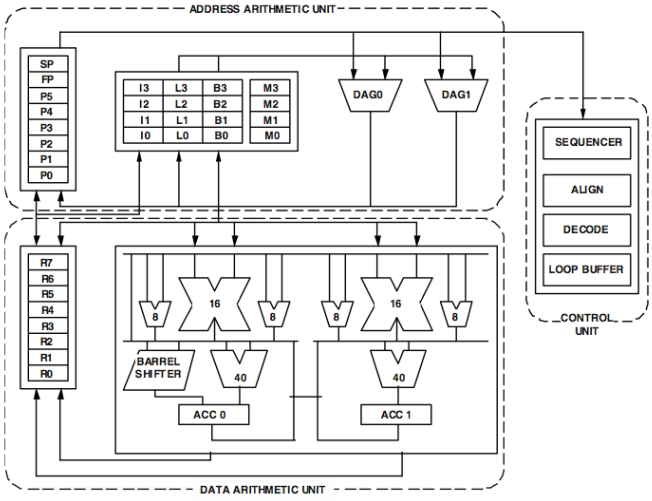

Fig. 2. The Blackfin processor core.

A. Blackfin BF561

The Blackfin derivative used here is the dual-core Blackfin BF561, which is included on the BF561 EZ-KIT LITE evaluation board. The BF561 [29] has the following memory available:

- 64 MB SDRAM main memory in 4 banks of 16MB (4x16MB), operating at up to 133 MHz (system clock)
- 128 KB on-chip L2 (8x16KB), at 300 MHz ($\frac{1}{2}$ core clock)
- 100 KB in-core L1, at 600 MHz (core clock), split into:
  - 32 KB instruction (16 KB SRAM; 16KB configurable as cache or SRAM)
  - 64 KB data (32 KB SRAM; 32 KB cache/SRAM)
  - 4 KB scratch-pad memory

V. Approaches to Using Memory

In our analysis we evaluate three different memory configurations: a worst-case baseline which maps both the original image and the processed image to SDRAM, and two which use low-latency memory. The first low-latency approach copies a partition of the original image to SRAM for each processing kernel, and saves the processed image again to SRAM, which is then written back to SDRAM. The second approach streams the processed subimages between kernels - only reading data from SDRAM when compulsory and writing results to SDRAM when the processing is complete. The following subsections explain these approaches more fully.

A. Per-Kernel Storage

This approach copies subimages into input records, runs a computational kernel, saves results into output records, and saves these records back to SDRAM; this process is then repeated for the next kernel in the control flow of the program. This approach is summarized in the following algorithm:

PK_Transform
1 **foreach** $kernel_i$ **in** Control Flow
2 **do foreach** $SUBIMAGE$ **in** $IMAGE$
3 **do** Copy($SUBIMAGE, kernel_i.in$)
4 Run($kernel_i(kernel_i.in, rows, cols, kernel_i.out)$)
5 Copy($kernel_i.out, SUBIMAGE$)

This approach is similar to performing source transformations of inner-most loops to maximize data locality. Such transformations may cause intra-kernel locality, but inter-kernel communication uses SDRAM, and incurs a per-kernel latency penalty. A side effect of this approach is that the results of each kernel can be saved, but at the cost of increased memory accesses for each kernel.

B. Streaming Local Stores

An alternative approach is to use the stream model. Under this model, SDRAM is only accessed for compulsory reads or completion writes (i.e., only after all n kernels in the control flow have been processed). During the processing of the image, the data *streams* from one kernel to the next, all in low-latency memory. This approach is summarized in the following algorithm:

Stream_Transform
1 **foreach** $SUBIMAGE$ **in** $IMAGE$
2 **do** Copy($SUBIMAGE, kernel_1.in$)
3 **foreach** $kernel_i$ **in** Control Flow
4 **do** Run($kernel_i(kernel_i.in, rows, cols, kernel_i.out)$)
5 Copy($kernel_n.out, SUBIMAGE$)

Unlike the previous approach, the inter-kernel communication does not use SDRAM and the results of each kernel are not saved - only the final results are saved. For programs where intermediate results are not important, the reduced number of memory accesses provides higher performance.

C. Developing Other Stream Image Processing Programs

Because our approach uses convolution kernel routines, other image processing algorithms (sharpening, embossing, etc.) could be implemented following the same methodology. In order to add more kernels to the stream, we utilized a C data structure representing a kernel, which included pointers to the input/output records and a callback for the kernel's function, and three routines shown below:

- addKernel - adds a kernel data structure to a linked list representing the control flow of the program
- newStreamRec - allocates low-latency memory for input/output records of a kernel
- mapStream - maps existing records for a kernel

An initialization using the above routines sets up the control flow of a stream program. Following this initialization, a list traversal of the control flow would call each kernel in the sequence specified. The addition of new kernels or stream mappings only requires a different initialization, without requiring a specialized stream compiler.

VI. Edge Detection Workload

A common problem for automotive computer vision systems is to highlight edges. For example, systems using multiple cameras to perceive road traffic often use software to create a map based on the edge detection of stereo images. This data can be used for lane guidance, vehicle tracking, automated braking control, etc.

Revisiting the generic IMAGEPROCKERNEL algorithm, our program uses 2D convolution as the TRANSFORM subroutine. During convolution, each pixel in the input image is transformed into a weighted sum of its neighbors. The weights are stored in a matrix usually referred to as the "kernel." To avoid confusion with the concept of computational kernels, we will refer to this matrix as a kernel-matrix. The similarity of terms is intentional: kernel-matrices should be stored to low-latency memory because a compute kernel frequently accesses its elements.

Our program performs edge-detection using two convolutions. The first performs a Gaussian blur to reduce noise (such as dust particles in an image), the second performs a Sobel gradient response operation in which the intensity of the resulting image is highest near edges (in either direction). Figure 3 shows the dataflow of this program.

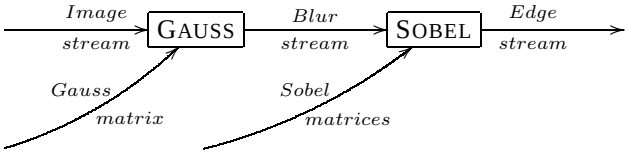

Fig. 3. Dataflow graph for Edge Detection program

To build a stream control flow for the program depicted in Fig. 3, the following initialization calls would be made:

ADDKERNEL(GAUSS)
ADDKERNEL(SOBEL)
NEWSTREAMREC(GAUSS, rows, cols, INPUT)
NEWSTREAMREC(GAUSS, rows, cols, OUTPUT)
NEWSTREAMREC(SOBEL, rows, cols, OUTPUT)
MAPSTREAM(GAUSS, OUPUT, SOBEL, INPUT)

VII. Implementation and Results

We implemented an edge detector in C using the Visual DSP++ 4.0 integrated development and debugging environment, and the web tutorial provided by [30] to convolve bitmap (BMP) source images with the kernel-matrices below: $\mathbf{Gauss}$ for blurring [31], and $\mathbf{Sobel_x}$, $\mathbf{Sobel_y}$ for vertical and horizontal edge detection [32].

$$\mathbf{Gauss} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\mathbf{Sobel_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{Sobel_y} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

To make use of the Blackfin's dual-MAC DSP instruction set, we used and adapted the Sobel assembly provided in the Blackfin SDK 2.0 [33]. For the Gaussian blur kernel, two pixels were processed concurrently; for the Sobel kernel, both horizontal and vertical convolutions were performed in parallel.

For edge detection, image boundaries are important. Using a partitioned image introduces erroneous edges due to the borders of the resulting sub-images. Both convolutions used 3x3 kernel-matrices, so each partitioned subimage needed to access the last row of the previous subimage. A moving window of "live" data was used; data was copied into SRAM, processed, saved, and the window re-addressed to repeat the process for the next sub-image.

Our input dataset is a 1920x1080 pixel (WxH) image extracted from a HD video. Below we describe the memory configuration used and the results obtained (presented in Tables I-IV. All results are generated by running on the real (versus simulated) Blackfin hardware and are the average of five trials per configuration.

A. Baseline Approach

The baseline represents the worst-case because the memory accesses are all mapped to SDRAM, as depicted in the following algorithm:

EDGEDETECT
1 RUN(GAUSS($origImage, rows, cols, blurImage$))
2 RUN(SOBEL($blurImage, rows, cols, edgeImage$))

$origImage$, $blurImage$, and $edgeImage$ are pointers to main memory. Table I shows the baseline results: despite the use of both MAC units and hand-coded assembly, the total runtime is reduced from 3.41 to 2.48 sec - only a 27% reduction. This result demonstrates that utilizing the memory hierarchy effectively is, for our image processing workload, more important than effectively using the computational resources.

TABLE I
BASELINE: WORST-CASE MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	0 (0)	0 (0)
Gauss convolution	954, 852, 092 (1.59)	568, 957, 265 (0.95)
Sobel convolution	1, 092, 902, 630 (1.82)	921, 313, 359 (1.54)

B. Per-Kernel Approach

Expanding on the algorithm PK_TRANSFORM, the per-kernel approach utilizes the BF561 by mapping each kernel's input and output records to L2 or L1 SRAM.

PK_ EDGEDETECT
1 **foreach** $SUBIMAGE$ **in** $IMAGE$
2 **do** COPY($SUBIMAGE$, GAUSS.in)
3 RUN(GAUSS(GAUSS.in, $rows$, $cols$, GAUSS.out))
4 COPY(GAUSS.out, $SUBIMAGE$)
5 **foreach** $SUBIMAGE$ **in** $IMAGE$
6 **do** COPY($SUBIMAGE$, SOBEL.in)
7 RUN(SOBEL(SOBEL.in, $rows$, $cols$, SOBEL.out))
8 COPY(SOBEL.out, $SUBIMAGE$)

In Table II, the effect of using L2 is shown. This approach introduces a delay due to copying image data into and out of L2, but shows that localizing data accesses impacts total runtime that is less than the Sobel convolution alone, in the worst-case. Also, the effect of using both MAC units is more pronounced, the total runtime is decreased from 1.77 to 0.84 sec - a reduction of 53%, almost double the baseline reduction.

TABLE II
PER-KERNEL L2 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	257, 576, 841 (0.43)	127, 812, 414 (0.21)
Gauss convolution	353, 170, 263 (0.59)	174, 660, 214 (0.29)
Sobel convolution	450, 098, 388 (0.75)	198, 909, 374 (0.33)

Table III shows that even though the copy time is about the same ($\sim$0.20 sec for the assembly implementation) due to the SDRAM latency, the convolution runtimes can be reduced with L1 accesses at core-clock speed.

TABLE III
PER-KERNEL L1 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	233, 324, 154 (0.39)	112, 447, 043 (0.19)
Gauss convolution	198, 205, 503 (0.33)	26, 998, 728 (0.04)
Sobel convolution	300, 954, 794 (0.50)	48, 514, 428 (0.08)

C. Stream Approach

Expanding on the algorithm STREAM_TRANSFORM, the stream and per-kernel approaches both map each kernel's input and output records to L2 or L1 SRAM. But the stream approach only accesses SDRAM when compulsory or when data has been completely processed and is ready to be saved.

STREAM_ EDGEDETECT
1 **foreach** $SUBIMAGE$ **in** $IMAGE$
2 **do** COPY($SUBIMAGE$, GAUSS.in)
3 RUN(GAUSS(GAUSS.in, $cols$, GAUSS.out))
4 RUN(SOBEL(GAUSS.out, $rows$, $cols$, SOBEL.out))
5 COPY(SOBEL, $SUBIMAGE$)

In Table IV the effects of using the stream model are shown. This approach reduces the copy delay because only compulsory or complete accesses occur. The convolution runtimes remain about the same compared to the per-kernel approach, but the overall runtime is reduced.

TABLE IV
STREAM L2 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	128, 774, 780 (0.21)	63, 907, 125 (0.11)
Gauss convolution	349, 059, 870 (0.58)	174, 687, 337 (0.29)
Sobel convolution	450, 126, 228 (0.75)	198, 936, 082 (0.33)

Table V shows that again the stream model provides the best runtime, with the convolution runtimes being about the same as in the per-kernel approach. In the next subsection we compare the approaches and summarize the results.

TABLE V
STREAM L1 MEMORY USAGE

	C code Cycles (Runtime, sec)	ASM code Cycles (Runtime, sec)
Memory copies	116, 647, 351 (0.19)	56, 225, 653 (0.09)
Gauss convolution	194, 094, 567 (0.32)	27, 028, 002 (0.05)
Sobel convolution	300, 982, 290 (0.50)	48, 540, 970 (0.08)

D. Summary

Tables VI and VII show the combined results for the three approaches, ordered by decreasing execution time in seconds, for both the C and assembly implementations.

TABLE VI
COMPARING MEMORY UTILIZATION APPROACHES WITH C IMPLEMENTATION.

	Total Runtime, sec	Speedup compared to Baseline
Baseline	3.41	1.00
Per-Kernel L2	1.77	1.93
Stream L2	1.55	2.20
Per-Kernel L1	1.22	2.80
Stream L1	1.02	3.34

TABLE VII
COMPARING MEMORY UTILIZATION APPROACHES WITH ASSEMBLY IMPLEMENTATION.

	Total Runtime, sec	Speedup compared to Baseline
Baseline	2.48	1.00
Per-Kernel L2	0.84	2.95
Stream L2	0.73	3.40
Per-Kernel L1	0.31	8.00
Stream L1	0.22	11.3

Accessing both MAC units shows little performance benefit for the baseline approach using only SDRAM. The total runtime is reduced from 3.41 to 2.48 sec, a mere 27% decrease attained by using hand-coded assembly. The penalty for using

higher latency memory outweighs the benefits of using the computational resources efficiently.

However when L2 is used, the assembly's power is shown as the edge detector runs over 2X faster for both low-latency approaches (and either implementation). Using L1 is even more beneficial, providing over 3X faster runtimes. Even higher performance is achieved by using the stream approach which uses low-latency meory as often as possible.

VIII. Related Work

Previous work used the Blackfin's configurable memory focused on mapping code to SRAM and a code layout tool [34]. Recently, frameworks to develop optimized media applications on the Blackfin have described a set of "templates" which exploit predictable data access patterns to make use of low-latency memory. [35], [36], [37]

The stream processing paradigm [38] is naturally suited to media applications [39], [40], which display large amounts of parallelism, little reuse of data, and a high ratio of computations to memory accesses. Some scientific applications have similar characteristics but a difference lies in memory access patterns. Whereas media applications fit well into streams because the gathering of data will be more or less sequential, scientific applications often need more random access to memory.

Despite this difference, Gummaraju and Rosenblum [41] found a 27% performance increase using the stream paradigm for certain scientific applications. The architecture used was the Intel Pentium 4 and the performance was limited due to the cache overhead of non-temporal loads and stores. The Blackfin has a configurable memory hierarchy and stands to benefit by using low-latency memory, without the overhead of maintaining the cache.

The Data Transfer and Storage Exploration methodology [11] performs data-dependence analysis and performs source-to-source code transformations (such as loop transformations) to ensure optimizations for the memory hierarhcy (for example, reorganizing inner and outer loops to improve producer-consumer locality). The stream model incorporates some of the same lessons of the DTSE project and its programming style might serve to reduce some of the loop transformations.

IX. Conclusion

In this paper, we examined the use of the configurable memory hierarchy on the Blackfin BF561, the use of the two MAC units for image processing by matrix convolution, and the application of the stream model to use low-latency SRAM.

We found that for both L2 and L1 SRAM, using the stream model in which accesses to SDRAM are limited to compulsory and completed reads and writes, respectively, can achieve speedups of 2.8X to 3.3X versus a worst-case baseline that uses SDRAM exclusively using C; and speedups of 3.4X to 11.3X using hand-optimized assembly code available from ADI.

The stream model discussed above provides a natural means to express image processing programs such that the memory hierarchy is effectively used (i.e., on-chip memory is used for the majority of data streams). The model is simple enough to be implemented using basic C data structures and callbacks to represent control flow, but powerful enough to achieve an order of magnitude speedup compared to the worst-case baseline approach.

X. Future Work

The Stream Virtual Machine [23] described above contains master control and slave kernel processors, and DMA units. The Blackfin supports dual cores and a DMA engine. If a low-level compiler for the SVM were developed to support it, then programs written in stream languages could potentially perform better on the Blackfin. Other embedded systems supporting similar features could also stand to benefit from the use of the stream model.

Acknowledgment

The authors would like to thank Mimi Pichey, Giuseppe Olivadoti, Richard Gentile, and Ken Butler from Analog Devices for their generous donation of Blackfin EZ-KIT Lite evaluation boards, software, and extender boards, and for their support of this project. We also extend our gratitude to the anonymous reviewers for their comments and suggestions.

References

[1] Wm. A. Wulf and Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. SIGARCH Computer Architecture News, 23(1):2024, 1995.

[2] Sally A. McKee. *Reflections on the Memory Wall*. In Proceedings of the 1st Conference on Computing Frontiers (CF '04), page 162, New York, NY, USA, 2004.

[3] Rita Cucchiara, Massimo Piccardi, and Andrea Pati. *Exploiting Cache in Multimedia* In Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS '99), Florence, Italy, 1999.

[4] Andrea Pati. *Exploring Multimedia Applications Locality to Improve Cache Performance* In Proceedings of 8th ACM International Conference on Multimedia (MULTIMEDIA '00), pp. 509-510. Marina del Rey, California, USA, 2000.

[5] Afrin Naz, Krishna Kavi, Philip Sweany, and Mehran Rezaei. *A Study of Separate Array and Scalar Caches*, In Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS'04), pp. 157-164, Winnipeg, Manitoba, Canada, 2004.

[6] Afrin Naz, Mehran Rezaei, Krishna Kavi, and Philip Sweany. *Improving Data Cache Performance with Integrated Use of Split Caches, Victim Cache and Stream Buffers*, In Proceedings of the 2004 Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA-2004), Antibes Juan-les-Pins, France, 2004.

[7] Nimrod Megiddo, and Dharmendra Modha. *Outperforming LRU with Adaptive Replacement Cache Algorithm*, Computer, 37(4):58-65, 2004.

[8] Nimrod Megiddo, and Dharmendra Modha. *ARC: A Self-Tuning, Low Overhead Replacement Cache*, In Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03), pp. 115-130, San Francisco, CA, USA, 2003.

[9] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. *Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems* In Proceedings of the Tenth International Symposium on Hardware/Software Codesign (CODES 2002), pp.73-78, Estes Park, Colorado, USA, 2002.

[10] Monica Lam and Michael Wolf. *A Data Locality Optimizing Algorithm* In Proceedings of the 12th Conference on Programming Language Design and Implementation (PLDI 1991), pp. 30-44, Toronto, Ontario, CA, 1991.

[11] Francky Cathoor, Koen Danckaert, Sven Wuytack, and Nikil D. Dutt. *Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors* IEEE Design&Test of Computers 18(3):70-82, 2001.

[12] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick . *A Case for Intelligent RAM* IEEEmicro 17(2):34-44, 1997.

[13] Sunaga, T, Peter M. Kogge, et al, *A Processor In Memory Chip for Massively Parallel Embedded Applications*, IEEE Journal of Solid State Circuits, Oct. 1996, pp. 1556-1559.

[14] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. *Stream Processors: Programmability and Efficiency* ACM Queue, 2(1):52-62, 2004.

[15] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. *Programmable Stream Processors*, ACM Computer 26(8):54-62, 20003.

[16] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. *Imagine: Media Processing with Streams*. IEEE Micro, 21(2):35-46, 2001.

[17] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. *Evaluating the Imagine Stream Architecture*. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA 2004), pp. 14-26, Munchen, Germany, 2004.

[18] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. *Merrimac: Supercomputing with Streams*, Proceedings of the ACM/IEEE conference on Supercomputing (SC '03) p. 35, Phoenix, Arizona, USA, 2003.

[19] Stream Processors, Inc., 455 DeGuigne Drive Sunnyvale, CA 94085, USA. *Stream Processing: Enabling a new class of easy to use, high-performance parallel DSPs*, revision 1.9, 2007, Document: SPI_MWP.

[20] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. *TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP* ACM Trans. on Architecture and Code Optimization, 1(1):62-93, 2004.

[21] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Sama Amarasinghe, and Anant Agarwal. *Bringing It All to Software: Raw Machines*, IEEE Computer, 30(9):86-93, 1997.

[22] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. *A Stream Compiler for Communication-Exposed Architectures*. SIGPLAN Not. 37(10):291-303, 2002.

[23] *The Stream Virtual Machine*. Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis, and Mark Horowitz. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT'2004), pp. 267-277, Antibes Juan-les-Pins, France, 2004.

[24] Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French. *A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels*, In Proceedings of the 30th International Symposium on Computer Architecture (ISCA 2003), pp. 410-421, San Diego, CA, USA, 2003.

[25] Sourav Chatterji, Manikandan Narayanan, Jason Duell, and Leonid Oliker. *Performance evaluation of two emerging media processors: VIRAM and Imagine* In Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS 2003), pp. 229.1, Nice, France, 2003.

[26] Bart Kienhuis and Ed F. Deprettere. *Modeling Stream-Based Applications Using the SBF Model of Computation*, Journal of VLSI Signal Processing Systems, July 2003, pp. 291-300.

[27] *Micro Signal Architecture from ADI and Intel*. `www.intel.com/design/msa/highlights.pdf`,

[28] Analog Devices, Inc., One Technology Way, Norwood, MA 02062, USA. *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*, revision 1.0 edition, June 2005, Part Number 82-000556-01.

[29] Analog Devices, Inc., One Technology Way, Norwood, MA 02062. *ADSP-BF561 Blackfin Processor Hardware Reference*, revision 1.0 edition, July 2005, Part Number 82-000561-01.

[30] Bill Green. *Edge Detection Tutorial*. `www.pages.drexel.edu/~weg22/edge.html`, 200 2.

[31] *Efficient algorithm for Gaussian blur using finite-state machines* Frederick M. Waltz and John W. V. Miller, Proc. SPIE Int. Soc. Opt. Eng. 3521, 3 34-341, 1998.

[32] Bob Fisher, Simon Perkins, Ashley Walker, and Erik Wolfart. *Sobel Edge Detector* as part of Univ. of Edinburgh HyperMedia Image Proces sing Reference, `www.cee.hw.ac.uk/hipr/html/sobel.html`, 1994.

[33] Analog Devices, Inc. *Software Development Kit (SDK) Downloads*, `www.analog.com/processors/platforms/sdk.html`, SDK Rel 2.00, January 2007.

[34] Kaushal Shanghai, David Kaeli, and Richard Gentile. *Code and Data Partitioning on the Blackfin 561 Dual-Core Platform*. In Proceedings of the 3rd Workshop on Optimizations for DSP and Embedded Systems, pp. 92-100, San Jose, CA, 2005.

[35] Kunal Singh and Ramesh Babu. *Video Framework Considerations for Image Processing on Blackfin Processors*. Analog Devices Inc., App Note EE-276, Rev 1, Sept 2005.

[36] Kaushal Sanghai. *Video Templates for Developing Multimedia Applications on Blackfin Processors*. Analog Devices Inc., App Note EE-301, Rev 1, Sept 2006.

[37] Glen Ouellette, and Kaushal Sanghai. *Efficient Data Management Frameworks for Multimedia Applications*, DSP-FPGA.com, Dec 2006. `Www.dsp-fpga.com/articles/ouellette_and_sanghai/`

[38] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo Lopez-Lagunas, Peter R. Mattson, and John D. Owens. *A Bandwidth-Efficient Architecture for Media Processing*. In Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 31), pp. 3-13, Los Alamitos, CA, USA, 1998.

[39] Ruby B. Lee and Michael D. Smith. *Guest Editor's Introduction: Media Processing: A New Design Target*. IEEE Micro, 16(4):6-9, 1996.

[40] Keith Diefendorff and Pradeep K. Dubey. *How Multimedia Workloads Will Change Processor Design*. Computer, 30(9):43-45, 1997.

[41] Jayanth Gummara ju and Mendel Rosenblum. *Stream Programming on General-Purpose Processors*. In Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38), pp. 343-354, Washington, DC, USA, 2005.

An ILP Formulation for Recomputation Based SPM Management for Embedded CMPs *

Hakduran Koc, Ehat Ercanli
Department of EECS
Syracuse University
Syracuse, NY, USA
{hkoc,eercanli}@ecs.syr.edu

Mahmut T. Kandemir, Ozcan Ozturk
Department of CSE
The Pennsylvania State University
University Park, PA, USA
{kandemir,ozturk}@cse.psu.edu

ABSTRACT

Both chip multiprocessors and scratch pad memories are being increasingly employed in embedded computing systems. At the confluence of these trends lies the problem of optimizing performance for chip multiprocessors equipped with on-chip SPMs. As against the most of the existing SPM management techniques, we propose an approach that reduces the number of off-chip memory accesses for data by recomputing the value of that off-chip data using the values of the data residing in on-chip SPMs. In this paper, we propose an ILP (integer linear programming) based approach driven by profile data to 1) decide the contents of on-chip SPMs and 2) make recomputation decisions to improve performance of the application being optimized. The experiments with six benchmark codes show that our proposed approach brings nearly 12.6% reduction in overall execution time.

1. INTRODUCTION

CMPs (chip multiprocessors) are becoming increasingly popular as they promise power efficiency and high performance through exploitation of high level parallelism in applications. In the embedded computing domain, one can speed up loop/data-intensive applications significantly by distributing loop iterations across multiple processors and choreographing data accesses carefully with the goal of optimizing locality. One of the critical problems for embedded systems using CMPs is to minimize the number of off-chip memory accesses for data. This is because such accesses are very costly in terms of both execution cycles and power consumption. There exist several approaches proposed in literature to reduce the number of off-chip memory accesses, including code and data optimizations and architectural enhancements.

Concurrent with the trend toward CMPs, we also witness a trend toward increasing use of software managed on-chip memories such as SPMs (scratch-pad memories). In order to minimize the number of off-chip accesses in an SPM based CMP, we need to maximize the SPM utilization as much as possible. That is, most of data requests should be satisfied from on-chip SPM memories, instead of going off-chip.

In this paper, we propose an approach which is significantly different than existing approaches. As against many of the existing SPM optimization schemes, we propose to use *recomputation* of data for cutting the number of off-chip data accesses. More specifically, instead of making an off-chip memory access, we recompute, if possible, the value of the requested off-chip data element using the on-chip data available in SPMs. Clearly, this can only be done if the data required for recomputing that off-chip data are available in on-chip SPMs. In addition, whether this can really bring any benefits or not depends on whether the cost of accessing these on-chip data elements plus the cost of recomputation is less than the cost of the off-chip access in question.

An interesting point here is that, since on-chip SPM accesses are much less expensive than off-chip memory accesses, we can afford to perform several on-chip accesses and still have less overall cost than an off-chip access. In this paper, we propose an ILP (integer linear programming) based approach driven by profile data to 1) decide the contents of on-chip SPMs and 2) make recomputation decisions to improve performance of the application being optimized.

We present not just the details of our ILP based formulation but also a set of experimental results that demonstrate the viability of our approach. The experiments with six benchmark codes show that our proposed approach brings nearly 12.6% reduction in overall execution time. Our experiments also show that the achieved performance improvements are consistent under the different values of some of experimental parameters.

The rest of this paper is structured as follows. Related work is discussed in Section 2. High level view of our proposed approach and the target architecture is explained in Section 3. Section 4 describes the data recomputation using an example. Details of the ILP based formulation are presented in Section 5, and an experimental evaluation of it is given in Section 6. Finally, the paper is concluded with a summary of our major contributions in Section 7.

2. RELATED WORK

On-chip memory components such as caches, scratch-pad memories, and on-chip DRAMs have been focus of much research, especially from both the performance power perspectives in embedded computing systems. Among them, Scratch-Pad Memory (SPM) has gathered much attention recently, due to its advantages over other on-chip memory components. These advantages include power/energy efficiency, reduced cost, better performance, and real-time predictability [4, 3, 21, 25, 8]. SPM is a software-managed (compiler or application) on-chip SRAM with guaranteed fast access time. Banakar et al [4] compare the conventional cache and SPM based organizations for the computationally intensive embedded applications. They report that SPM occupies, on average, 34% less area; consumes 40% less energy; and achieves 18% reduction in execution time. On the other hand, Panda et al [22] report that on-chip caches using static SRAM can consume 25% to 45% of total chip power. Keitel-Schulz and Wehn [13] report that on-chip memory occupies more than 50% of total chip area. Given these benefits, SPM has been utilized as an alternative (Motorola's M-core, 68HC12 and TI's TMS370Cx7x) to the conventional on-chip cache memories or together with the cache memories (ARM10E

*This work is supported in part by NSF Career Award 0093082.

and ColdFire MCF5) in several embedded designs. SPM has also been utilized in chip multiprocessors such as TI's TNETV3010 [2], 250MHz [17] and 600MHz [6] CMPs. Avissar et al [3] have projected that memory systems without caches will continue to grow in embedded systems.

In order to utilize the limited on-chip memory space, researchers proposed several data allocation and SPM management strategies. We can group the SPM management strategies into two major categories: static and dynamic. Panda et al [21] propose a static data management scheme. In this work, the data partitioning remains valid throughout the execution. They first assign all constant and scalar variables to SRAM, and the arrays that do not fit in SPM to DRAM. Then, they fill SPM with the remaining arrays based on their life times and access frequencies. On the other hand, [26] proposed a dynamic memory allocation scheme. In this method, the compiler analyzes the application and inserts additional code to copy select variables into the SPM. In [25], Steinke et al investigate another SPM allocation method to improve the energy consumption. The two techniques above consider both program code and application data. Kandemir et al [12] propose another dynamic SPM management scheme for application data. Their scheme takes into account the data layout and data access patterns; and transfers the data tiles (some portion of array data) among memory components during the course of execution. Liet et al [19] introduce another SPM management scheme, called memory coloring. The authors transform the SPM management problem into one that can be solved using existing graph coloring algorithms. Arrays to be placed in the SPM are clustered into equivalent array classes. Even though the approach is promising for automating SPM placement problem, it increases, when splitting arrays by adding temporary array variables, memory space consumption. Verma et al [27] propose three allocation strategies, namely Non-Saving, Saving and Hybrid, to improve the energy consumption of multiple processes sharing the same scratch pad memory space. The first strategy divides the SPM into disjoint regions and each process is allowed to use at most one region. In the second strategy, SPM is shared among all processes of the application and the hybrid is the combination of first two.

In addition, SPM management has been studied in the context of chip multiprocessors. [18, 24, 23] studied chip multi-processor architectures. Kandemir et al [10] present a compiler strategy to reduce off-chip memory accesses. The proposed approach reduces energy-delay product by up to 33.8% by improving the reuse of the on-chip data. Ozturk et al [20] study another on-chip memory management scheme in the context of chip multiprocessors. They propose a compiler-directed ILP strategy for application specific on-chip memories. Avissar et al [3] targeted the memory systems without caches and proposed an elegant compiler strategy for the management of on- and off-chip heterogeneous memory systems.

The idea of recomputing a value instead of fetching it from the memory is utilized in the context of register allocation. Briggs et al [5] propose a strategy to rematerialize (recompute) a register value when it is cheaper than storing and retrieving it from the memory. However, data recomputation in the context of embedded systems has not been studied much. [16] proposed an approach to improve the memory space requirements of the applications represented by task graphs. Kandemir et al [9] studied the storage-recomputation tradeoffs in memory-constrained embedded processing in order to reduce memory space consumption. Koc et al used data recomputation in the context of uniprocessor systems to improve the energy consumption of multi-bank memory systems [15] and to reduce the memory space requirements of data-intensive applications [14]. Unlike our data recomputation based SPM placement strategy for CMP based platforms, these studies targeted the single processor systems.

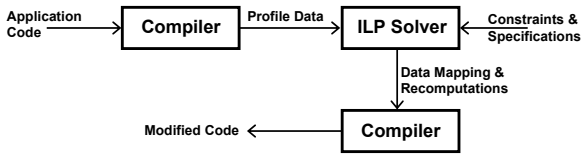

Figure 1: A High level view of the on-chip data recomputation based approach.

3. HIGH LEVEL VIEW OF THE APPROACH AND TARGET ARCHITECTURE

The high level view of the approach presented in this paper is given in Figure 1. Given the source code, the application is first divided into *phases* and *profiled* by the compiler. A phase is considered as the part of the source code in which data access pattern changes are minimal. This is very important for the performance of any embedded architecture with software-managed on-chip memories since the execution latency of the data-intensive applications greatly influenced by the available on-chip data. A phase is defined in terms of loop nests in our context (since we focus on array-based applications). At a very extreme case, the entire program can be considered as one phase. In addition, the phase concept enables us to incorporate the dynamic aspect of SPM management into our model. The advantages of the dynamic SPM management (vs. static one) are emphasized by previous research as explained in Section 2. Also, profiling gives us the data blocks considering all program data accessed by each processor. A *data block* (*tile*) is the chunk of data that can be moved across the different memories under the compiler guidance during the course of execution.

Then, this information is passed to the ILP solver in order to find the most efficient on/off-chip memory mapping and the available data recomputations (which will be explained in detail in Section 4). Considering the state-of-the-art SPM management, our SPM placement strategy basically favors the data blocks used for data recomputation to be on-chip over the other data tiles. The ILP solver gives us, as output, the most efficient SPM/off-chip memory mapping and the recomputation candidates for each phase of the execution. Recall that, our goal is to reduce the number of off-chip memory accesses by recomputing the value of the desired data elements instead of fetching them from the off-chip memory. Next, using the output of the solver, the compiler modifies the program code to implement the identified data recomputations and inserts explicit data transfer calls between the phases of the application code.

In this paper, the target embedded architecture is a chip multiprocessor similar to [3], which does not support a hardware-controlled cache but utilize software-managed on-chip memory. On the other hand, the SPM management strategy employed in this system is a dynamic one similar to [12]. As shown in Figure 2, the architectural model consists of a CMP and an off-chip main memory. The CMP contains four homogeneous processor cores, four equal-size SPMs (each directly connected to one processor), and a circuitry that manages the synchronization and communication among these components. Note that, throughout the paper, the SPM directly connected to a processor is called as the processor's local SPM and the others as the remote (neighboring) SPMs. The possible memory access types are also illustrated in Figure 2, using curved arrows. A processor can access to its local SPM (P4-SPM4), to

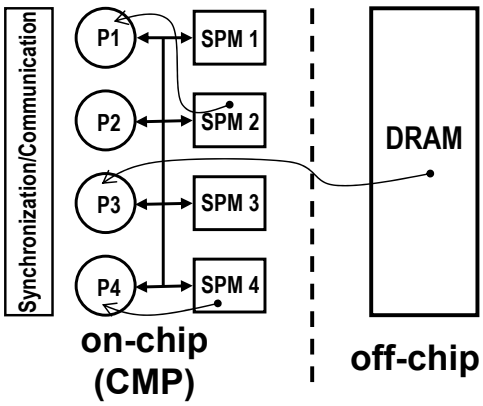

Figure 2: Target system architecture (four processors, their local SPMs, and an off-chip memory) and the possible memory access types: local (P4-SPM4), remote (P1-SPM2), and off-chip (P3-DRAM).

a neighboring (remote) SPM (P1-SPM2), or the off-chip memory (P3-DRAM). A data access for a processor to its local SPM takes a small latency, while an access to a remote on-chip memory requires relatively longer time. On the other hand, the main memory is an off-chip DRAM with much higher access latency. Memory address space is partitioned among the on-chip SPMs and the off-chip memory. During the course of execution, the application data may be placed into any part of the address spaces and can move back and forth among the on/off-chip memory components under compiler control. The data transfers take place at phase boundaries and the unit for data transfer is a tile, as mentioned earlier. In this target architecture, an access to the SPMs is through the on-chip interconnects, while an access to the off-chip memory occurs over external buses.

In this paper, our main focus is to formulate the on-off chip memory mapping (and the corresponding data recomputations) in an integer linear programming environment and present the experimental evaluation. The details of the code modifications carried out by the compiler are out of the scope of this paper. We also do not study the optimum phase partitioning (our work can work with any phase partitioning; clearly, a good phase partitioning will lead to better savings). Still, considering the significant changes in data access pattern occur across the loop nests, our phase partitioning is pretty reasonable. Note that, the data transfers between the memory components during execution implements the dynamic SPM management.

4. AN EXAMPLE USAGE OF DATA RECOMPUTATION

In embedded processing platforms the efficient utilization of limited on-chip memory space is a crucial issue from the performance point of view. On-chip memory management strategy becomes more important especially for the systems executing data-intensive embedded applications. In such systems, the execution latency of the application is almost proportional to the number of off-chip memory accesses. In this paper, we propose an efficient SPM management strategy based on on-chip data recomputations to cut the number of off-chip memory accesses.

The data recomputation approach is based on recomputing the value of an off-chip data, required by the current computation, using the available on-chip data elements instead of accessing the off-chip memory and fetching it from there, if doing so is beneficial in terms of performance. We basically substitute a memory access with recomputation. In some cases, the recomputation of an off-chip data may require more than one on-chip memory access. This is not a problem from performance point of view since an off-chip access is typically much costlier than an on-chip access.

The embedded systems with software-managed on-chip SPMs provide a suitable platform for data recomputation. This is because, in such systems, the compiler has the opportunity of placing the preferred data (in our case, the data to be used for recomputation) into on-chip SPMs and keeping it there during a period of execution. In this work, we target data-intensive applications, which frequently occur in image/video processing domain. These applications usually consist of loops in which multi-dimensional arrays are manipulated and the total data space is dominated by these array variables. In other words, the scalars and the application code occupies a small portion of the total data space. Consequently, we focus on array variables, and assume that the scalar data are stored in register files or in some reserved area in one of the on-chip SPMs.

```
int H[J], K[J], L[J], M[J], N[J];
for (i=1 to J, i++)

   K[i] = M[i] + N[i];        // S1
   L[i] = K[i]/2 + M[i] - 1;  // S2
   H[i] = 2 * K[i] + N;       // S3

end for
```

(a)

```
for(i=0 to J, i++)
   K[i] = M[i] + N[i];
   L[i] = (M[i] + N[i])/2 + M[i] - 1;
   H[i] = 2 * (M[i] + N[i]) + N[i];
end for
```

(b)

Figure 3: Example code fragments: a) Array declarations and original code, b) Modified code fragment after computing K[] explicitly.

Now let us consider the simple example in Figure 3 to demonstrate the on-chip data recomputation. Using the example code fragments, we compare the conventional SPM placement strategies and our recomputation based approach. Figure 3.a gives the original code fragment with array declarations. The example code fragment has only one loop nest (i.e. there is one phase to be considered during the course of execution). Let us assume, for simplicity, that J is 100; data arrays are one-dimensional; and arrays M[] and N[] are initialized earlier in the program. We assume the architecture shown in Section 3. Local, remote, and off-chip memory access costs are assumed to be 2, 5, and 80 clock cycles, respectively. Also, it is assumed that addition, subtraction, multiplication, and division operations take 1, 1, 4, and 4 clock cycles, respectively. In the source code, there are five arrays, each occupying 400B of data. Hence, the total data space for array variables is 2000B (400×5). Assuming that each SPM is capable of holding 200B of data, the total on-chip memory space is 800B (200×4). Each array is assumed to be partitioned into four data blocks of equal sizes (100B). More specifically, K[] contains the blocks b0 through b3; M[] b4 through b7; N[] b8 through b11; L[] b12 through b15; and H[] b16 through b19. The data blocks for arrays K[], M[], and N[] are shown in

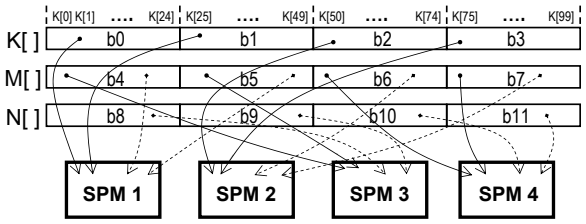

Figure 4: Data arrays and the corresponding blocks mapped into SPMs. The solid curves represent the SPM placement in the original code, while the dashed represents the placement in the modified code.

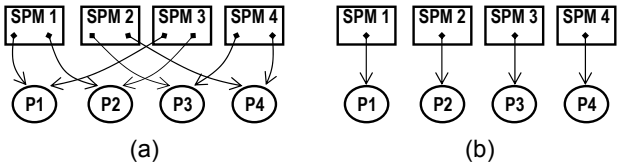

Figure 5: The SPM accesses of the processors during data recomputation. a) Random SPM mapping for recomputation, b) Processor-aware SPM for recomputation.

Figure 4. The partitionings for L[] and H[] are similar. Note that, even though only two blocks can be mapped into single SPM in our example, in realistic scenarios many data blocks may reside in on-chip memories.

First, let us estimate the execution latency of the given code fragment using the conventional SPM placement techniques. The current techniques place the most frequently accessed blocks into the on-chip SPMs in order to minimize the number of off-chip accesses and improve the performance. The access frequencies of the blocks in the original code are as follows: b0 through b3 are accessed 75 times, b4 through b11 50 times; and others 25 times. Since only eight blocks can be placed into on-chip, the compiler selects b0-b3 and four of b4-b11 to be mapped to SPMs. Let us assume that the blocks b0-b1, b2-b3, b4-b5, and b6-b7 are mapped to SPM1, SPM2, SPM3, and SPM4, respectively, as illustrated in Figure 4, using the solid curves. Please note that, in this case the compiler has the flexibility of selecting the data blocks to be on-chip (any four of b4-b11). In such a case, our placement strategy favors the data blocks that are used for possible recomputations to be on-chip. Since there is no dependency in this code, we can assume that the iterations 1-25, 26-50, 51-75, and 76-100 are assigned to the processors P1, P2, P3, and P4, respectively, in our target architecture. Given this setup, the execution latency of the original code can be estimated as 8925 clock cycles. Note that, our approach can be applied to the loops in which inter-iteration dependences exist since the present dependences do not affect the on-chip data recomputation opportunities.

We now discuss how our recomputation based SPM placement strategy works for this code fragment. An analysis of the original code fragment in Figure 3.a indicates that the references of array K[] is computed in statement S1, and then, used in statements S2 and S3. This brings us the opportunity of computing the array reference K[i] in statements S2 and S3 using the corresponding references to arrays M[] and N[]. In order to implement this recomputation, our placement strategy should meet two conditions: 1) the data block that holds the reference K[i] should reside in off-chip memory, and 2) the data blocks that contains M[i] and N[i] is to be mapped into one of the on-chip SPMs. Consequently, our SPM management strategy forces M[i] and N[i] to be on-chip and K[i] to be off-chip if this is beneficial in term of performance. Keeping in mind these two conditions, let us implement the recomputations by substituting the left hand side of S1 (K[i]) by the right hand side (M[i]+N[i]) in statements S2 and S3. The resulting modified code is given in Figure 3.b. Note that, the access frequencies of the data blocks have changed after this modification. More specifically, blocks b4-b11 are accessed 100 times and others 25 times. Using the new access frequencies, blocks b4-b11 are placed into on-chip SPMs. For now, let us assume that b4-b5 is mapped to SPM1; b6-b7 to SMP2; b8-b9 to SPM3; and b10-b11 to SPM4 as illustrated in Figure 4. The curved lines picture the new SPM placement. Based on the new memory mapping and the assumptions given above, the new execution latency is estimated to be 7050 clock cycles in the target architecture. As can be clearly seen, the proposed data recomputation based SPM placement cuts the execution latency up to 21% by reducing the off-chip memory accesses, for one iteration, from 4 to 3 in the modified code.

Note that, given the new SPM placement above, processor 1 (P1) needs to access its local SPM (SPM1) and a remote SPM (SPM3) to be able to perform the recomputations (i.e., K[1]=M[1]+N[1]); and, similarly, P2 accesses two remote SPMs (SPM1 and SPM3) to carry out the recomputations (i.e., K[30]=M[30]+N[30]) as shown in Figure 5.a. These remote SPM accesses increases the execution latency of the recomputation since a remote memory access is costlier than a local memory access (i.e, 5 vs. 2 clock cycles in our target architecture). Now, let us consider another placement scenario for the blocks chosen to be on-chip (b4-b11). Given the recomputation candidate (K[i]=M[i]+N[i]), we see that in order to recompute the first 25 references of K[i], a processor needs to access the first 25 references of M[i] and N[i]. In other words, P1 needs to access to b4 and b8 in order to update b0. Similarly, P2 accesses to b5 and b9; P3 to b6 and b10; and P4 to b7 and b11 in order to perform the recomputations. To reduce the recomputation cost, the blocks used for the recomputation of a reference should be mapped to the same SPM as much as possible. For example, the recomputation cost for K[30] is 11 clock cycles using the previous mapping (2 remote SPM accesses plus addition). However, it is just 5 cycles if b5 and b9 reside in P2's local SPM (2 local SPM accesses plus addition). Consequently, blocks b4-b8, b5-b9, b6-b10, and b7-b11 are to be mapped into SPM1, SPM2, SPM3, and SPM4, respectively. In this processor-aware SPM mapping for the blocks chosen to be on-chip, each processor accesses only its local SPM for recomputation as shown in Figure 5.b. Based on this new SPM mapping, the execution latency is estimated to be 6750 clock cycles. This is an additional 4.3% improvement in the performance over the previous mapping. This clearly indicates the importance of the processor-aware SPM mapping of the data blocks chosen to be on-chip.

We need to emphasize that the simple example above is given for illustrative purposes to explain the data recomputation and present an understanding of some requirements encountered during the ILP formulation. Also, recomputation cost and SPM placement is simplified. The cost functions and SPM placement performed by our ILP model is given in detail in the next section.

In performing the SPM placement, the strategy presented here favors the data blocks that are used for recomputation to be on-chip considering state-of-the-art SPM management strategies. The basic idea is to map the data blocks that are frequently accessed and used for recomputation to the on-chip memory; and the blocks whose references could be recomputed to the off-chip memory. In doing

so, we aim at recomputing the value of the off-chip data using the available on-chip components and reducing the number of off-chip memory accesses to improve the performance of the application. One interesting point of the approach is that the recomputations performed in the original code change the access frequencies of the data blocks in favor of our data recomputation based SPM placement strategy. In other words, recomputations increases the access frequencies of the data blocks that are used for recomputation and decreases the frequencies of the blocks whose references could be recomputed. Consequently, this improves the performance by cutting the number of off-chip memory accesses.

As stated earlier, each processor in our target architecture can access its local SPM, any of the remote SPMs, and the off-chip memory, with the access latencies of C_{local}, C_{remote} and $C_{offchip}$, respectively ($C_{local} < C_{remote} < C_{offchip}$). In some cases, recomputing the value of off-chip data element may require more than one on-chip memory access (local or remote). Then, the cost of recomputation becomes the cost of memory accesses plus the operation costs. The condition that makes the data recomputation preferable is that the cost of on-chip data recomputation should be less than the off-chip memory access latency, $C_{offchip}$. Note that, in general, the off-chip memory access cost, $C_{offchip}$, is much higher than both C_{local} and C_{remote}.

5. PROBLEM FORMULATION

In this section, we discuss how ILP can be used for determining optimum on/off-chip memory mapping for the recomputation based approach. We use a commercial ILP solver Xpress-MP [1, 7] to formulate and solve the proposed data recomputation based SPM placement problem. Table 1 gives the constant terms used in our ILP formulation and their definitions. Note that, as shown in Section 3, the number of on-chip memory components (SPMs) is equal to the number of processors. Hence, the total on-chip memory size is $N_P \times S_{SPM}$ and the off-chip memory size is assumed to be unlimited. In our model, we consider five different operations, namely, negation, addition, subtraction, multiplication, and division. Given the source code, the profile data provides the number of accesses to block *bl* on the right hand side of statement *st* of phase *ph*, $N_{b\_right}(ph, st, bl)$; the number of occurrences of operation *o* in statement *st* of phase *ph*, $N_{op}(ph, st, o)$; and the block on left hand side of statement *st* of phase *ph*, $B_{left}(ph, st)$, as input to the ILP solver.

Constant	Definition
N_P	Number of processors
N_{ph}	Number of phases in the program
N_{block}	Number of blocks in entire data-set
$N_{st}(ph)$	Number of statements at phase *ph*
S_{SPM}	Size of each SPM
S_{block}	Block size
C_{local}	Cost of local memory access
C_{remote}	Cost of remote memory access
$C_{offchip}$	Cost of off-chip memory access
$C_{op}(i)$	Execution cost of operation *i*

Table 1: The constant terms used in our ILP formulation and their definitions. All costs are in terms of execution cycles.

In order to formulate our problem, the following decision variables are used in the ILP model. We use 0-1 variable map to specify where a data block is mapped in a certain phase of the execution of the program. Remember that in our target architecture, there are four on-chip SPMs (each is connected to a processor) and an off-chip memory. More specifically,

- $map_{ph,bl,mem}$: indicates whether the block *bl* is mapped to the memory component mem at phase ph.

Similarly, $recomp$ is used to specify the required data recomputation at different phases of the execution.

- $recomp_{ph,S_i,S_j}$: indicates whether the left hand side of statement S_i is recomputed in statement S_j at phase ph.

The execution cost of each statement in the program code is captured by integer variable exe_time. Specifically,

- $exe\_time_{ph,st}$: indicates the execution cost of statement st at phase ph.

Given the decision variables, we first discuss the conditions related to on-chip data size and memory mapping. In our performance oriented model, an important constraint to be met is to limit the number of blocks mapped on on-chip SPMs. More specifically, the total data size of the blocks mapped on an SPM in any phase of the program can not exceed the SPM size, S_{SPM}. In mathematical terms, we have

$$\sum_{i=1}^{N_{block}} S_{block} \times map_{ph,i,mem} \leq S_{SPM}, \quad \forall ph, mem. \quad (1)$$

Recall that the SPMs are of the same size. Similarly, the total on-chip data size should be less than or equal to the total on-chip memory space at a phase.

$$\sum_{i=1}^{N_{block}} \sum_{j=1}^{N_P} S_{block} \times map_{ph,i,j} \leq N_P \times S_{SPM}, \quad \forall ph. \quad (2)$$

In our target memory architecture, the address space is partitioned into on-chip SPM space and the off-chip memory. As a result, a block can be mapped either on one of the SPMs or on the off-chip memory in a phase. This condition can be written as follows:

$$\sum_{i=1}^{N_P+1} map_{ph,bl,i} = 1, \quad \forall ph, bl. \quad (3)$$

Note that, the total number of memory components is $N_P + 1$ (including the off-chip memory) as used in the above formula.

Next, we present the conditions to be satisfied by an array reference to be eligible for recomputing its value. The recomputation of an array reference that resides in off-chip memory is carried out by using the available on-chip data instead of fetching it from the off-chip memory. This brings us two conditions to be able to perform a recomputation. First, the data block that contains the array reference considered for recomputation should reside in the off-chip memory. More specifically, the array reference on the left hand side of S_i of ph must be mapped to the off-chip memory. Second, all data blocks that are used for recomputation should reside in the on-chip memory space. In other words, the data blocks that contain the array references on the right hand side of statement S_i of phase ph should be mapped on one of the on-chip SPMs. Otherwise, there is no need for recomputation since the recomputation will not be beneficial. These conditions can be captured as follows:

$$map_{ph,bl,N_P+1} = 1, \quad \forall ph, bl \text{ s.t. } bl = B_{left}(ph, S_i). \quad (4)$$

$$\sum_{i=1}^{N_P} map_{ph,bl,i} \;=\; 1, \quad \forall ph, bl \text{ s.t. } N_{b\_right}(ph, S_i, bl) \neq 0. \tag{5}$$

Note that, in the above equation, the $(N_P + 1)$th memory component corresponds to the off-chip memory. On the other hand, the condition that makes the data recomputation effective (and valid) from the performance point of view is that the recomputation cost, $C_{recomp}(ph, S_i)$, of a statement S_i at phase ph should be less than the off-chip memory access cost $C_{offchip}$. That is,

$$C_{recom}(ph, S_i) < C_{offchip}, \quad \forall ph, S_i. \tag{6}$$

The recomputation cost has two components (the cost of operations in the statement and the access cost of data blocks on the right hand side) and can be defined explicitly as follows:

$$C_{recomp}(ph, S_i) = \sum_{j=1}^{N_{block}} \sum_{k=1}^{N_P+1} N_{b\_right}(ph, S_i, j) \times map_{ph,j,k} \times C_{access}(ph, j, pr), \quad \forall ph, S_i, pr. \tag{7}$$

In the above equation, $C_{access}(ph, bl, pr)$ indicates the cost of an access to block bl by processor pr. The cost could be C_{local}, C_{remote} or $C_{offchip}$.

We now discuss the conditions required to be able to recompute an array reference on the left hand side of statement S_i in statement S_j at phase ph. S_j should contain the array reference on the right hand side and should occur later in the code. Thus, the following condition must hold true:

$$N_{b\_right}(ph, S_j, B_{left}(ph, S_i)) \neq 0 \text{ and } i < j, \; \forall ph, S_i, S_j. \tag{8}$$

However, in some cases, an array element can be updated more than once within a given phase. In that case, care should be taken to recompute the reference using the latest statement. Specifically, if the left hand side of S_i is to be recomputed in S_j, the left hand side of S_i should not be updated by S_k between S_i and S_j.

$$\neg(B_{left}(ph, S_k) \neq B_{left}(ph, S_i) \text{ and } i < k < j), \; \forall ph, S_i, S_j, S_k. \tag{9}$$

If the conditions in Equations (4), (5), (6), (8), and (9) hold, then the decision variable $recomp(ph, S_i, S_j)$ is set to 1. That is to say, the left hand side of S_i is recomputed on the right hand side of S_j at phase ph.

$$recomp(ph, S_i, S_j) = 1, \quad \forall ph, S_i, S_j. \tag{10}$$

If a valid recomputation is available, then the following variables are updated.

$$\begin{aligned} N_{b\_right}(S_j, bl) \;+\!=\; & N_{b\_right}(S_j, bl) \times N_{b\_right}(S_i, bl) \\ N_{op}(S_j, o) \;=\; & N_{op}(S_j, o) + N_{b\_right}(S_j, bl) \times N_{op}(S_i, o) \\ N_{b\_right}(S_j, B_{left}(ph, S_i)) \;=\; & 0, \quad \forall S_i, S_j, bl, o. \end{aligned} \tag{11}$$

Next, we relate the decision variables, map and $recomp$, presented above. To be able to recompute a target array reference, certain blocks should reside in the on-chip memory space. More specifically, if a block is updated by a recomputation, the blocks that contain the references on the right hand side must be mapped to one of the SPMs. That is,

$$\sum_{i=1}^{N_P} map(ph, bl, i) \geq recomp(ph, S_i, S_j), \quad \forall ph, S_i, S_j, bl \text{ s.t. } N_{b\_right}(ph, S_i, bl) > 0. \tag{12}$$

Note that, since a block can be mapped to only one of the SPMs, the statement on the left side of expression 12 could be at most 1.

The execution cost of each statement in the program code can be estimated as follows:

$$exe\_time(ph, S) = \sum_{i=1}^{N_{block}} \sum_{j=1}^{N_P+1} N_{b\_total}(ph, S, i) \times map_{ph,i,j} \times C_{access}(ph, i, pr), \quad \forall ph, S, pr. \tag{13}$$

In the above equation, $N_{b\_total}(ph, st, bl)$ represents the total number of access to block bl including ones on the left hand side.

Finally, since our approach aims at reducing the total execution cost of a given application, the objective function in our ILP model can easily be derived by adding exe_time of each statement in the application as following:

$$\min \quad \sum_{i=1}^{N_{ph}} \sum_{j=1}^{N_{st}(i)} exe\_time(i, j). \tag{14}$$

Please note that we do not include the cost of block transfers across the phases during the execution since the block transfers has no effect on SPM mapping or recomputation. Also, previous research [11] shows that the dynamic management of SPM mapping gives better results than the static management does.

6. EXPERIMENTAL EVALUATION

6.1 Setup

In this section, we present an experimental evaluation of the proposed recomputation based SPM management strategy. In order to show the effectiveness of the approach, we used six data-intensive benchmarks from the Livermore, Perfect Club and SPEC benchmark suites. Some of their important characteristics of the benchmark programs are given in Table 2. The target applications for the proposed strategy are the data-intensive, array-dominated applications, which are heavily used in the image/video processing domain. These benchmarks are written in C and consist of loop nests using which one- to three-dimensional data arrays are manipulated. The total data sizes of the benchmarks range from 122KB to 281KB. The first two columns in Table 2 give the name of the benchmark and the suite from which the benchmark is taken. The third column shows the dataset sizes for the benchmarks.

As stated earlier, to formulate the recomputation based SPM placement strategy and determine the memory mapping of the data blocks and the possible recomputations, we use a commercial ILP solver, Xpress-MP [1]. Our target architecture is similar to one described in Section 3 and simulated using a custom simulator. The simulated architecture is a four processors CPM in which each processor is a pipelined, out-of-order embedded CPU. Each processor has its 10KB local SPM. The data block sizes are chosen to hold at least 10 data blocks in an SPM. On the other hand, the size of the off-chip DRAM was kept large enough in order to hold the entire data sets required by the applications. In the simulator, accesses to the local SPM, remote SPMs, and the off-chip DRAM are assumed to take 2, 5, and 80 clock cycles, respectively.

6.2 Results

The main experimental results are given in the last three columns of Table 2. The fourth column gives the original execution time of the benchmark codes without applying our data recomputation based SPM placement strategy. Note that, the strategy employed during these experiments is a dynamic state-of-the-art SPM placement

Benchmark Name	Source	Data Size (KB)	Orig. Exec. Time (msec)	New Exec. Time (msec)	Performance Gain (%)
adi	Livermore	241.2	70.2	67.2	4.3
apsi	SPEC	122.4	40.7	37.0	9.1
bmcm	Perfect Club	123.2	457.2	350.2	23.4
eflux	Perfect Club	275.4	93.8	80.4	14.6
tomcatv	SPEC	280.8	106.5	98.1	7.9
wss	SPEC	122.8	459.0	384.2	16.3

Table 2: The benchmarks used for experimental evaluation, their important characteristics, and performance improvements.

strategy based on [12]. The fifth column in the table presents the execution latency of the benchmarks after the proposed ILP based strategy is applied. Finally, the last column reports the performance improvements. The performance gain brought by our approach over the conventional SPM placement strategy is 12.6%, on the average. We see that the improvements brought by our approach vary between 4.3% and 23.4%. The performance gain for the adi benchmark is small (4.3%). Even though the SPM mapping found by our ILP based approach enables the on-chip recomputations, the recomputation costs are relatively high for this benchmark in our four-processor CMP. On the other hand, our strategy achieves up to 23.4% improvement for bmcm benchmark. This is due to two reasons: first, the application code exhibits more recomputation opportunities against other benchmark codes, and second, the SPM placement is a very effective one for the available recomputations. In these experiments, we see around 4% increase in the code size. When we consider the target data-intensive applications in which the source code occupies quite small portion of total (code, array, scalar) data space, the increase in code size is not significant.

For sensitivity analysis, we first vary the available SPM size in our next set of experiments. Please note that the base configuration contains four processors with their local SPMs of equal size (10KB). Hence, the total on-chip memory size is 40KB (4×10). In these experiments, the SPM sizes considered are 5KB, 10KB, 15KB and 20KB. The normalized performance improvements are given in Figure 6 for the benchmarks in our experimental suite. Please note that, increasing the on-chip memory size decreases the execution time of an application. However, these results are especially important in the sense that there are possible recomputations even for different sizes of on-chip memory; and our placement strategy finds an efficient SPM mapping to reduce the number of off-chip accesses. In the figure, the performance gains for apsi and wss are zero when the SPM size is 20KB. This is because, in this configuration, about 2/3rd of the total application data reside in on-chip memories, as a result, there is no effective recomputation opportunity for those benchmark.

Next, we change the number of processors keeping the on-chip SPM space constant. In other words, the total on-chip memory space (40KB) in the base configuration is distributed equally to the available processors. Figure 7 presents the performance gains for different number of processors varying from 2 to 16. As can be seen from the figure, the configuration with two processors gives the best performance results. This is mainly due to the reduction in recomputation cost. More specifically, in this case, the processors have larger local SPMs (compared to other configurations) and need to access the remote SPMs less frequently. On the other hand, when the number of processors is increased, the performance gains decrease since a processor needs to access the remote SPMs more frequently. This, in turn, increases recomputation cost and eventually the total execution latency. As we could expect, the change in

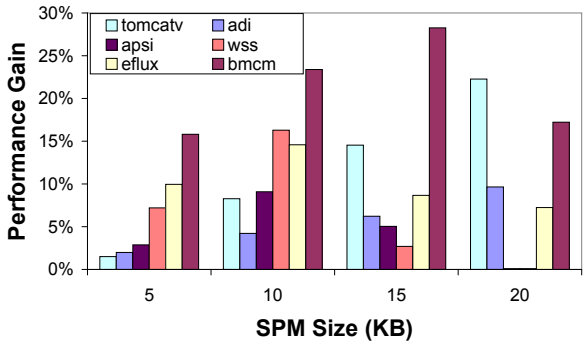

Figure 6: Normalized performance improvements when the SPM size is changed.

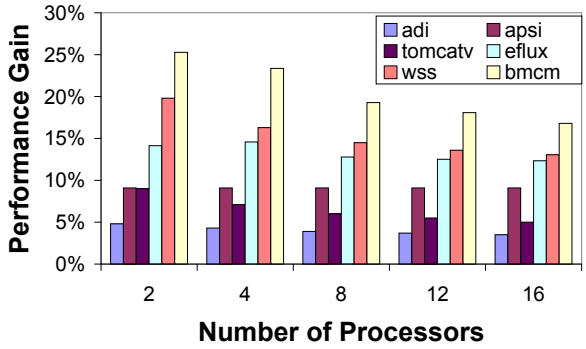

Figure 7: Normalized performance improvement when the number of processors is varied.

execution time for bmcm is large since the processors require frequent accesses to the neighboring SPMs to perform the recomputations. One interesting result in this figure is that the performance gain for apsi is around 9.1% for diffrent number of processors. This indicates that the required recomputations are mostly carried out using the processors' local SPMs.

In our final set of experiments, we changed the memory access latencies to see the effect of different recomputation costs in our ILP placement strategy. Recall that, in our base configuration, the memory access latencies are C_{local}=2, C_{remote}=5, and $C_{offchip}$=80. The evaluated configurations are as follows: Configuration 1: C_{local} =2, C_{remote} =20, $C_{offchip}$=80 and Configuration 2: C_{local}=1, C_{remote}=3, $C_{offchip}$=80. The results are given in Figure 8 for benchmarks eflux and wss. As can be seen from the figure, when the remote SPM access latency is increased (Configuration 1), the performance gains for both benchmarks decrease since the recomputation cost becomes larger. On the other hand, when the on-chip memory (local or remote) access costs are reduced, we see an in-

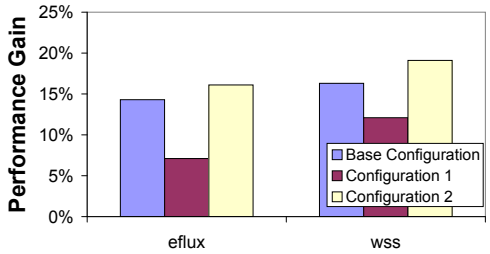

Figure 8: Performance gains with the different memory access latencies. Base configuration: C_{local}=2, C_{remote}=5, $C_{offchip}$= 80; Configuration1: C_{local}=2, C_{remote}=20, $C_{offchip}$= 80; and Configuration2: C_{local}=1, C_{remote}=3, $C_{offchip}$= 80.

crease in the performance gains for both benchmarks since the recomputation cost becomes less significant.

7. CONCLUSIONS

In this paper, we have presented a recomputation based SPM management strategy in an ILP environment for embedded CMPs and presented an experimental evaluation of the proposed scheme using six data-intensive embedded benchmarks. The data recomputation approach computes the value of the off-chip data, required by the current computation, using on-chip (SPM resident) data elements, instead of fetching the off-chip data from the memory. The proposed SPM management strategy modifies the SPM mapping by favoring the data blocks that are used for recomputation to be on-chip in order to reduce the number of off-chip memory accesses. The experimental results show the effectiveness of the proposed approach by reducing the overall execution latency by 12.6% over a state-of-the-art SPM management scheme.

8. REFERENCES

[1] Xpress-MP. *http://www.dashoptimization.com/home/downloads/pdf/mosel.pdf*, 2002.

[2] Tnetv3010 infrastructure VOP gateway solution, http://focus.ti.com.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of International Symposium on HW/SW Codesign*, Colorado, 2002.

[5] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *Proceedings of the Conference on Programming Language Design and Implementation*, San Francisco, California, USA, 1992.

[6] S. Kaneko et al. A 600mhz single-chip multiprocessor with 4.8gb/s internal shared pipelined bus and 512kb internal memory. *IEEE Journal of Solid-State Circuits*, 39(1):184–193, 2004.

[7] C. Gueret, C. Prins, M. Sevaux, and S. Heipcke. *Applications of optimization with Xpress-MP*. Dash Optimization, 2002.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[9] M. Kandemir, F. Li, G. Chen, G. Chen, and O. Ozturk. Studying storage-recomputation tradeoffs in memory-constrained embedded processing. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Munich, Germany, 2005.

[10] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the Conference on Design automation*, New Orleans, Louisiana, 2002.

[11] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Conference on Design automation*, pages 690–695, Las Vegas, Nevada, United States, 2001.

[12] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. A compiler based approach for dynamically managing scratch-pad memories in embedded systems. *IEEE Trans. on CAD*, 23(2):243–260, 2004.

[13] D. Keitel-Schulz and N. Wehn. Embedded dram development: Technology, physical design, and application issues. *IEEE Design and Test*, 18(3):7–15, 2001.

[14] H. Koc, E. Ercanli, M. Kandemir, and S. W. Son. Compiler-directed temporary array elimination. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems*, February 2006.

[15] H. Koc, O. Ozturk, M. Kandemir, S. H. K. Narayanan, and E. Ercanli. Minimizing energy consumption of banked memories using data recomputation. In *Proceedings of International Symposium on Low Power Electronics and Design*, Tegernsee, Germany, October 2006.

[16] H. Koc, S. Tosun, O. Ozturk, and M. Kandemir. Reducing memory requirements through task recomputation in embedded multi-cpu systems. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, April 2006.

[17] T. Koyama, K. Inoue, H. Hanaki, M. Yasue, and E. Iwata. A 250-mhz single-chip multiprocessor for audio and video signal processing. *IEEE Journal of Solid-State Circuits*, 36(11):1768–1774, Nov. 2001.

[18] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[19] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, Saint Louis, Missouri, 2005.

[20] O. Ozturk, G. Chen, M. Kandemir, and M. Karakoy. An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *Proceedings of the IEEE Annual Symposium on Emerging VLSI Tech. and Arch.*, Karlsruhe, Germany, 2006.

[21] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the European Conference on Design and Test*, Paris, France, 1997.

[22] P. R. Panda, A. Nicolau, and N. Dutt. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[23] S. Richardson. Mpoc: a chip multiprocessor for embedded systems. In *HP Laboratories Technical Report HPL-2002-186*, Palo Alto, CA, USA, 2002.

[24] S. F. Smith. Performance of a gals single-chip multiprocessor. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, 2004.

[25] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Paris, France, 2002.

[26] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, San Jose, California, 2003.

[27] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Proceedings of the Workshop on Embedded Systems for Real-Time Multimedia*, New York, NY, 2005.

A Code Layout Framework for Embedded Processors with Configurable Memory Hierarchy

Kaushal Sanghai Alex Raikman
Ken Butler
Analog Devices
Norwood, MA 02062, USA
kaushal.sanghai@analog.com
alex.raikman@analog.com
ken.butler@analog.com

David Kaeli
ECE Department
Northeastern University
Boston, MA 02115, USA
kaeli@ece.neu.edu

Abstract

Several embedded processors now support configurable memory hierarchies to exploit application specific workload characteristics. To take advantage of memory reconfigurability, automated software optimization techniques are generally lacking and application developers often resort to a hand tuned code layout. This not only increases the time to market of embedded products but may also result in an inefficient mapping.

To address this issue, we have developed a framework which incorporates profile guided code layout algorithms to efficiently map code on the available L1 code memory configurations. Given a L1 memory configuration, we show that the code layout problem can be mapped to a NP-complete Knapsack problem or to a combination of Knapsack problem and a graph coarsening problem.

In this paper we present the performance benefits of applying our framework to the multimedia applications from the EEMBC consumer benchmark suite. We evaluate the code layout algorithms on the ADI's Blackfin single core embedded processor.

1. Introduction

Configurable memory hierarchies are now commonly available in embedded processors. For example, the ADI's Blackfin family of processors [1] and LSI Logic's CW33000 Risc microprocessor core [2] support on-chip L1 memories which can be partitioned to behave as L1 SRAM and/or L1 Cache.

...

When the L1 memory is configured as L1 SRAM, the code is mapped statically on to the L1 memory. The code mapped to the L1 SRAM memory resides throughout the execution of the program and is never evicted from the L1 memory. This ensures that a core request to a L1 SRAM has guaranteed single cycle access and is not subject to a cache miss. A part of the L1 memory can be configured to behave as cache and this part of the L1 memory is termed as L1 Cache. In the case of L1 Cache, an instruction fetch may result in a cache miss and subsequently resulting in a core access to the external memory. The code which resides in the cache is dictated by the cache replacement policies. When the L1 memory is partitioned as L1 SRAM and L1 Cache, then we term it as L1 SRAM/Cache configuration.

Given such a memory subsystem, performance can be greatly increased by exploiting application specific workload characteristics. For example in case of L1 SRAM memory, one can map functions with real time criticality and most of the hot code sections in the L1 SRAM space. This eliminates most of the cache misses which in turn reduces the core access to the slower off-chip external memory. In effect, both, the average memory access time and the external memory bandwidth requirements are reduced. But sometimes not all of the critical code sections may be mapped in the available on-chip L1 SRAM space and therefore the remaining code sections have to be mapped to the off-chip external memory. To reduce the memory access latency in such cases a part of L1 SRAM can be configured to behave as cache.

To take advantage of a heterogeneous SRAM and cache memory architecture, automated software optimization techniques are generally lacking. Thereby, an application developer often resorts to a hand tuned code layout which not only increases the time to market of embedded products but may also result in an inefficient mapping. To address this issue, we have developed a framework which incorporates automated code layout techniques for the various L1 memory configurations. The framework uses the execution profile in-

formation and the L1 code memory configurations to produce an efficient code layout.

For just the cache memory, considerable amount of research has been done in the area of efficient code mapping [3, 4, 5, 6, 7]. Prior work [3] has developed code layout algorithms wherein, they map frequently called caller/callee functions in contiguous memory locations. More advanced algorithms for code reordering were developed by [4, 5, 6, 7]. For an L1 SRAM memory, we [8] mapped the code layout problem to a Knapsack problem [9]. Others have formulated the problem of mapping for SRAM memories as an Integer Linear Programming(ILP) problem [10, 11, 12, 13, 29].

For the heterogeneous SRAM and cache memory architecture we show that the code layout problem can be mapped to a combination of Knapsack and a graph coarsening problem. We also achieve better layout by ordering the code sections based on temporal reuse information.

For the L1 SRAM/Cache configuration we can partition the code and map part of the application code in the L1 SRAM memory and map the remaining in the external address space. To efficiently utilize both the SRAM and cache memory, we develop strategies such that they mutually benefit each other. For example,a cache memory can help to further save the L1 SRAM memory space. In the presence of cache, functions with high temporal locality which were originally mapped to L1 SRAM can be displaced to external memory. The functions with high temporal locality now mapped to the external memory would be friendly to cache and thus would not degrade the performance. Also, by placing hot functions and low temporal code in SRAM most of the cache conflicts are eliminated. Thus, for the remaining code that has to be mapped to the external memory very simple code layout techniques [3] for cache memories can be adopted.

We combine the code layout algorithms for the L1 SRAM configuration [8] and the L1 SRAM/Cache configuration in one common framework. We evaluate the code layout algorithms on six multimedia encoder/decoder algorithms from the EEMBC [14] consumer benchmark suite. In this work we use a ADI Blackfin 533 processor installed on a EZkit hardware board [1]. We show performance improvements ranging from 6% to 35% for different L1 SRAM/Cache configurations using our code layout algorithms.

The rest of the paper is organized as follows. In the next section we describe the memory architecture we are targeting. Section 3 presents the code layout algorithms for different L1 memory configurations. In section 4 we discuss the framework and its major components and section 5 describes the methodology of our evaluations. We present the results for the code layout algorithm for the L1 SRAM/Cache configuration in section 6 and some discussion in section 7. Finally, we discuss some related work and conclude in sections 8 and 9, respectively.

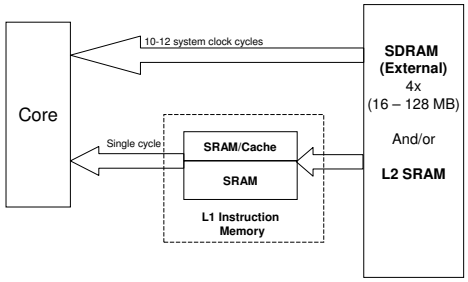

Figure 1. Instruction memory architecture for ADI's Blackfin family of processors.

2. Memory Architecture

In this section we will discuss the memory hierarchy model of the more recent embedded processors. We also describe the possible memory configurations and the tradeoffs involved in selecting a particular layout.

2.1 Memory model

A typical memory hierarchy model for some of the modern embedded processors is shown in figure 1. The on-chip L1 SRAM provides single cycle access to the core. Part of L1 SRAM can be configured to behave as cache. The L2 memory is optional and some processors may not implement it. If implemented, an L2 memory can be either configured as SRAM or cache. L2 access times are typically much longer than L1, but provide better performance than accessing off-chip SDRAM. For the rest of this paper we will omit L2 memory from the discussion, and focus on optimizations for cores with a single level of on-chip memory. The external memory is SDRAM and access to SDRAM generally results in a delay of several system clock cycles. A similar architecture for L1 data memory is discussed in [11] in more details.

2.2 Memory configurations

Given such a L1 memory model, we can have three possible configurations, although some processors could provide additional options.

1. **L1 SRAM:** In this configuration there is no cache and the entire L1 memory is available as SRAM memory. If the entire code fits in the L1 SRAM then this would provide the best performance. For code size greater than the L1 SRAM size, the code that spills over to the external memory will incur a delay of several system clock cycles if requested by core. For this configuration to be effective, only a very small number of code accesses should access external memory.
2. **L1 Cache:** This is similar to the L1 memory architecture prevalent in most general purpose microprocessors. All the code resides in the off-chip external memory and is cached in the L1 memory upon core request. The disad-

vantage of a cache memory is that it increases the *worst case access time* [15] which is critical for real time applications. It also increases the external bus bandwidth requirements which may prove costly for applications with streaming data buffers in external memory. Minimizing external bus bandwidth requirements is always a challenge in embedded systems. Also, performance may suffer if the application has poor locality behavior.

3. **L1 SRAM/Cache:** In this configuration most of the critical code sections can be mapped to the SRAM. The rest of the code which spills into external memory is cached. If most of the executed code can be placed in the L1 SRAM portion of the memory, then the majority of the compulsory and conflict misses are avoided. This should also reduce external bus bandwidth requirements. The cache can provide low latency access to infrequently executed code.

In the next section we describe the code mapping algorithms for the above L1 code memory configurations.

3. Code Mapping Algorithm

The problem of mapping is constrained to a combinatorial optimization problem based on the L1 memory configuration. In this section we discuss the code mapping algorithms for the L1 SRAM/Cache configurations in detail. The code mapping for L1 SRAM is described in [8] and L1 Cache configuration has been studied extensively in prior research and is not the targeted configuration for this work. Thus, we only summarize the algorithms and discuss only the relevance of L1 SRAM and L1 cache as they relate to a more heterogeneous L1 SRAM/Cache configuration.

3.1 L1 SRAM Code Mapping

For an efficient L1 SRAM code mapping, one approach would be to simply map the most frequently executed code (i.e., the hottest code) sections in L1 SRAM based only on their relative execution percentage. But the layout can be more effectively guided by also weighting the code sections by their corresponding binary size. This leads us to a very familiar NP complete Knapsack problem [9].

In the Knapsack problem every object is characterized by its value and weight. The objective is to maximize the sum of values of the objects in the Knapsack given a bound on the total weight of the Knapsack. This problem is also known as the 0/1 Knapsack problem. For the L1 SRAM code layout problem the goal is to maximize the execution percentage(value) of the code sections mapped to the L1 SRAM memory (Knapsack) given the constraint on the total size (weight) of the L1 SRAM memory.

If E_i is the execution percentage of each code section, and S_i is the size of the corresponding code section then the problem can be formulated as follows:

$$max\left(\sum_{i=1}^{n} E_i\right) \tag{1}$$

in the L1 memory, with the constraint that:

$$\sum_{i=1}^{n} S_i \leq \text{L1 memory} \tag{2}$$

The 0/1 Knapsack problem is a NP complete problem but by using greedy heuristics, near optimtal solutions can be obtained. We incorporate a simple greedy algorithm in the framework and it produces close to optimal solutions quickly.

Using the above equations, we use the execution percentage of each code section, code sizes, and the L1 memory size to drive the algorithm.

Greedy Algorithm: We compute the *hotness ratio* for a code section as its execution percentage in the overall program execution time over size. The code sections are sorted in decreasing order of hotness ratio. The sorted code sections are then added to the Knapsack until the L1 memory size bound is reached. As each new object is added, the L1 SRAM size is checked. If the L1 SRAM size is exceeded, the code section is not included and other objects are considered. In this way the algorithm obtains a solution as close as possible to the total L1 SRAM size bound.

3.2 L1 Cache Code Mapping

As discussed before, a L1 instruction cache memory is prevalent in most general purpose microprocessors and has been the focus of most prior work [3, 4, 5, 6, 7] in efficient code layout. In [3], Pettis and Hansen use the function caller/callee frequency count and a "closest is best" strategy to map code sections. By mapping caller/callee function pairs to contiguous memory locations, cache conflicts are reduced. The intuition behind this is that there is high temporal locality exhibited by caller/callee function pairs. Thereby, mapping caller/callee in contiguous memory locations in the external address space, a core fetch to those functions would result in them going to different cache lines and thus should reduce cache conflicts.

Subsequent techniques have used more advanced algorithms such as cache coloring [4], temporal reference graphs [7, 5], conflict miss graphs [6] for procedure reordering. For the memory model we are targeting most of the conflict misses are eliminated by mapping the most valuable code sections in L1 SRAM as determined by the L1 SRAM code layout algorithm. Thus, we do not need to incorporate more sophisticated algorithms for L1 cache layout.

3.3 L1 SRAM/Cache Code Mapping

In the L1 SRAM/Cache configuration we can partition the code sections to be mapped in the L1 SRAM and external memory. The code layout algorithm should aim to ef-

fectively utilize the L1 SRAM space, as well as produce an efficient layout for the L1 Cache memory.

To take advantage of L1 SRAM and cache memories we define the following three objectives the code layout algorithms should achieve

1. For the L1 SRAM memory the objective is to map the most valuable functions in L1 SRAM such that we maximize the execution percentage from L1 SRAM memory.
2. To place functions with low temporal locality in L1 SRAM and higher temporal locality functions (cache friendly) in external memory.
3. For the L1 cache memory, the objective is to minimize cache conflicts. Therefore the code layout algorithm should place functions in external memory efficiently.

In the next section we further describe how we could use the temporal reuse measure to further save the L1 SRAM space and use a similar technique developed in [3] to produce an efficient cache layout.

3.3.1 L1 SRAM Mapping

By using the execution percentage and size of the code section, we would be able to map most of the executed code in L1 SRAM. But by using the temporal reuse information of the code sections we can further save space in L1 SRAM memory by only mapping functions with poor temporal locality in L1 SRAM. This also ensures the cache performance would be efficient since only functions which exhibit good locality have been mapped to the external off-chip memory.

There could be many code sections that possess poor temporal locality. Therefore, it is important to consider their hotness ratio in order to prioritize code sections to be mapped in L1 SRAM. We measure the temporal locality of a function by computing its reuse distance [16]. If a function has a large reuse distance (i.e., poor temporal locality) then it is accessed less frequently. We use the reuse distance metric to guide code layout as follows:

If a function has a large reuse distance, it is likely that it will be evicted from the cache and so mapping it to external memory should not degrade the performance. Similarly, if a function has a small reuse distance, there is less of chance for it to be replaced in the cache. Thus, if a function has low temporal reuse and high hotness ratio it should have higher priority for its placement in L1 SRAM. We can use the cache for managing cache-friendly code.

Greedy Algorithm: Functions responsible for less than 1% of relative execution are pruned from the call graph. We sort the remaining functions by further dividing their hotness ratio by temporal reuse measure. Then we apply our greedy layout algorithm as described in section 3.1 to identify the code sections that will be placed in L1 SRAM.

3.3.2 L1 Cache mapping

The remaining code is mapped to the external off-chip memory and is subsequently cached in L1 cache. By using the function call graph, we can formulate code layout algorithm as an undirected graph coarsening problem with weighted edges and nodes. Every function is assigned a node and the caller/callee pair is connected via an edge. Edges are weighted by the call frequency of the corresponding caller/callee pair. The nodes are weighted by their binary size.

We map the most called caller/callee function in contiguous memory locations [3]. But instead of using just the "closest is best" strategy as is used in [3] we also take the cache configuration and the size of the functions as an input to our algorithm.

Greedy Algorithm: The input to the algorithm is the L1 cache configuration and the weighted call graph. To produce coarser sub graphs we merge connected nodes based on the relative edge weights. The algorithm can be briefly explained in the following three steps.

Step 1: Sort the edges of the call graph by the frequency count caller/callee pair (edge weight).

Step 2: Set the threshold on the maximum size of the merged node (coarser subgraph). This is equal to the cache line size (denoted as M_{th}).

Step 3: *For* all edges in the sorted list, start merging nodes. Given that the ith edge connects nodes A and B, and given that S_A and S_B denote the size of A and B, respectively, nodes that are grouped together are assigned a common merged node id.
Case 1: For both A and B, no merged ID is assigned and $S_A + S_B \leq M_{th}$. Merge nodes A and B by assigning them a common merged node id.
Case 2: A has already been assigned a merged node id (A is already part of a merged node).
If the total size of the merged node containing A would exceed M_{th} if B would be added,
then assign a new merged node id to B.
else merge B with the node containing A.
Case 3: Only B is assigned the id. This is the same as case 2, though just exchange A and B.
Case 4: Both A and B have assigned ids.
continue with the next edge;
end for;
end algorithm;

Thus, we map the functions as specified by the Knapsack problem to the L1 SRAM space, and the remaining functions are mapped to external memory based on the solution of the graph coarsening problem.

3.4 Conflicting optimization techniques

Compiler optimizations such as function inlining could also reduce the cache conflicts but it would also increase the

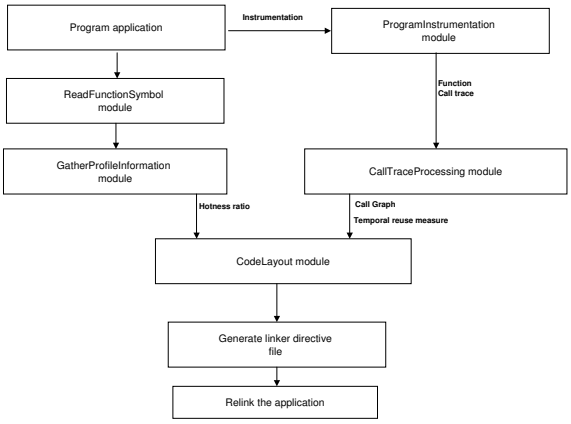

Figure 2. Framework for L1 code memory layout

code size. By changing only the mapping of the functions at link time, the code size remains the same. Also, statically analyzing the program behavior can help to improve code layout, though little control flow information is available at compile time [17]. Also hardware (i.e., cache organization) can be used to further reduce conflicts, though tends to be more appropriate for high performance versus embedded platforms.

4. The Framework

The previous sections described the code layout algorithms and the greedy algorithms for the available L1 memory configurations. The entire process from gathering profiles, selecting a particular layout algorithm to producing linker directive files is integrated in a framework. The framework allows for design space exploration to optimize the code memory layout for an embedded system implementation.

Figure 2 shows the major framework components and the process steps involved in producing the final layout. The current framework only supports mapping at a function granularity, but can be extended to finer granularities such as basic blocks. The framework starts with the ReadFunctionSymbol module. It reads the function symbol name, function size, function memory start address from the elf header embedded in the executable. The application is then executed on the target processor and the GatherProfileIinformation module stores the execution frequency of each function obtained from the profile run.

The ProgramInstrumentation module instruments the code at a function granularity to generate the function call graph. The collected call trace information is processed to build the call graph and provides temporal reuse information per function. Every function is assigned a node and the caller/callee frequency counts are represented by edge weights. It should be noted that if the user only wants an efficient L1 SRAM layout then the instrumentation and call graph construction steps can be skipped.

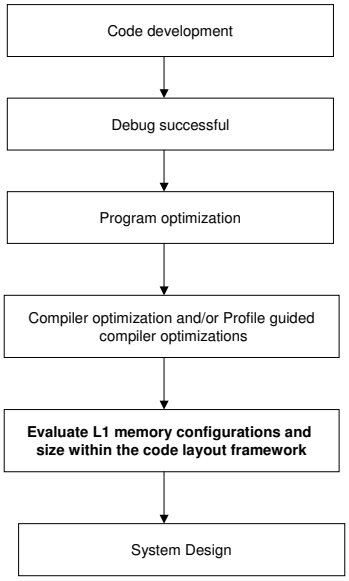

Figure 3. Framework usage in the overall embedded system implementation process.

The execution percentage and size of each function along with the call graph information (if instrumentation option is exercised), are then passed on to the CodeLayout module. The framework incorporates the algorithms for the two different L1 memory configuration options, L1 SRAM and L1 SRAM/Cache. We assume that a user would never select the L1 cache configuration since an L1 SRAM is always available for the processors targeted in this work.

A new set of directives are required to specify the memory mapping for each code section. The Output module produces a linker directive file where every function symbol name is labeled with a directive. This file is added to the project and the project is re-linked to produce an efficient layout.

To provide more information to the user, a plot of L1 memory size vs. the execution coverage out of L1 SRAM memory can also be displayed within the framework. Typically, different L1 memory sizes are available for a given family of processors and therefore an optimal L1 SRAM size could be determined from the graph.

Figure 3 shows where in the embedded system implementation, the framework can be exercised. It should be noted that if the sources or compiler switches are changed, then the entire process for code layout should be repeated. This is because optimization can affect the size of the code sections and the associated profile information.

5. Methodology

All the experiments are performed on ADI's Blackfin EZ-kit hardware board. VisualDSP++ 4.0 is used as the development environment for compiling, linking and building ex-

Benchmark	Code size(KB)	# of functions
JPEG2 encoder	56	380
JPEG2 decoder	61	388
MPEG2 encoder	84	330
MPEG2 decoder	68	351
MPEG4 encoder	197	480
MPEG4 decoder	131	404

Table 1. Code size (KB) and number of functions in each benchmark.

ecutables for the Blackfin processor. The tool provides a rich set of profiling tools and linking directives necessary for code mapping. The tool also provides a set of automation API's which help us to develop software utilities outside of the development environment (Visual DSP++) to extract information from the tool and process it automatically. These utilities are integrated within the framework and use the automation API's to collect the profiles and symbol information. They communicate to the VisualDSP++ tool via the Windows com interface.

Blackfin processors are designed specifically for consumer multimedia applications. Thus, we have evaluated our code layout algorithms on the EEMBC [14] consumer benchmark suite. If an application's code fits entirely in L1 SRAM, then mapping all of it to L1 SRAM would give the best performance. Therefore, from the benchmark suite we focus on applications which have large code sizes. We have selected the JPEG2, MPEG2 and MPEG4 encoder/decoder benchmark programs which have code size greater than 50KB. These benchmarks are standard video and image encoding/decoding algorithms widely used in multimedia applications.

Before using the framework, the programs are compiled with optimization turned on for maximum performance. The first step in the process is to collect the function symbol information by reading the elf header from the executable and obtaining the execution profile of the functions in the program by executing the program.

The instrumentation module is invoked to generate the call trace. The trace is post processed to produce the call graph. We use the standard inputs contained in the EEMBC benchmark suite for gathering the profile information.

We measured the performance in terms of the number of cycles consumed for program completion. Hardware performance counters available on the Blackfin processors are used to compute the cycle count.

To evaluate the code layout algorithms, we compare the performance for different L1 memory sizes. A part of L1 SRAM is used to map code sections (as determined from the code layout algorithms) and the rest can be used for mapping critical real time functions. In Blackfin, the instruction cache is a 16K 4-way set associative with a 32 bytes line size. It can be configured as 4K direct mapped or 8K 2-way set associative.

We choose a minimum of 8K L1 SRAM memory and a 4K direct mapped cache. We also perform our experiments with a 12K SRAM and 4K cache and a 8K SRAM and 8K cache. The above three L1 memory configurations are compared to the fully available 64K SRAM and 16K cache L1 memory on a Blackfin.

6. Results

Next, we present results for the 6 benchmark programs from the EEMBC consumer benchmark suite. Figure 4 shows the performance benefits obtained for the L1 SRAM/Cache configuration using our code mapping algorithm for the 6 different code layouts. A 12K (8K SRAM and 4K cache) of L1 memory space with no code layout optimizations is chosen as a baseline configuration. For all the six benchmarks the input used is the standard image provided in the EEMBC suite. Each data input comprised of at least fifteen CIF sized frames (352x240) for MPEG encoding or decoding, and a CIF sized image for JPEG encoding/decoding.

In the graph, the first bar shows the relative improvements after mapping the most executed (ME) code sections in the L1 SRAM space and using the default layout for the remaining code sections mapped to external memory. The next two bars show the performance benefits in using the Knapsack(KS) and the temporal reuse (TR) for the L1 SRAM along with the default linker layout for the functions in the external memory. The remaining 3 results are for the same L1 SRAM layout, but maps the remaining code sections using our graph coarsening technique (GC) with the help of a function call graph.

As can be seen in the figure, we achieve an average improvement of 19% for the KN-TR-GC algorithm (which combines the hotness ratio with temporal reuse) for the L1 SRAM layout and the graph coarsening for the external memory layout. While, most of the performance benefits are obtained by mapping the most frequently executed code in the L1 SRAM space, a further improvement of 4% is obtained by formulating the problem as a Knapsack problem. By incorporating temporal reuse information, a further improvement of 1% is possible for some of the benchmarks.

By efficiently mapping the code sections in the external memory, consistent improvement is seen over all the benchmarks. For the MPEG4 decoder, the benefits are as much as 7%. It should be noted that the performance improvements are more pronounced for the MPEG4 programs because of its larger code size as compared to the MPEG2 or JPEG2 programs.

In figure 5 we show the average performance improvement over all benchmarks for three different L1 SRAM/Cache memory size. The baseline is the default layout for the respective L1 memory configuration. The performance benefits for the 12K SRAM and 4K cache are slightly greater

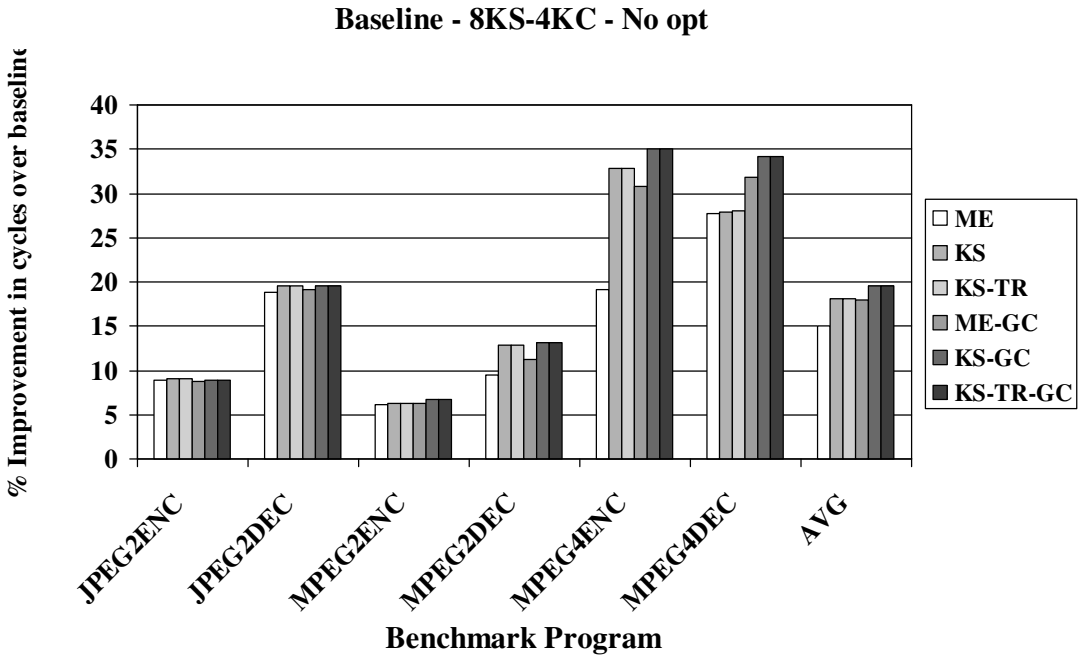

Figure 4. Figure shows the relative performance improvements for JPEG2, MPEG2 and MPEG4 encoder/decoder benchmark programs. A 8K SRAM and 4K Cache with no code layout optimization is taken as the baseline configuration. The relative performance improvements in terms of cycles over the baseline for 6 different layouts is shown. In the graph ME= Most Executed, KS=Knapsack,TR=Temporal Reuse and GC=Graph Coarsening.

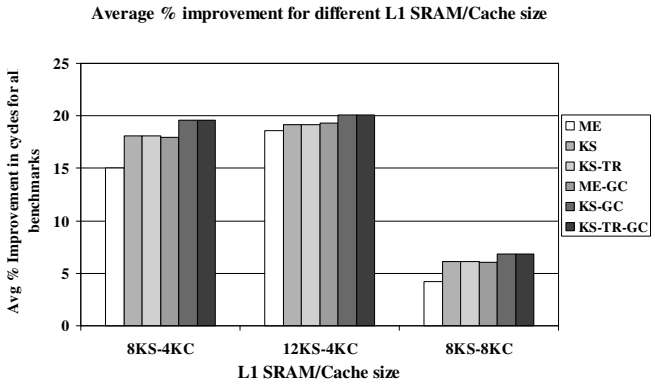

Figure 5. Figure shows the average performance improvements over all benchmarks for three different L1 SRAM/Cache memory size. In the graph ME= Most Executed, KS=Knapsack,TR=Temporal Reuse and GC=Graph Coarsening.

because the default layout does not make effective use of the increased L1 SRAM space. Also, in the case of 8K SRAM and 8K cache, the increased cache size exploits the locality even in the default layout and there is less than 7% benefit in using the code layout algorithms.

But as we can see for both the 12K SRAM/4K cache and 8K SRAM/8K cache L1 memory, the gains in using our advanced code layout techniques (i.e., KN, TR, or GP, as compared to ME) are diminishing. Most of the improvements are obtained by mapping just the hottest code sections or by applying the Knapsack algorithm. If most of the executed code can be mapped to the L1 SRAM space then there is little opportunity for improvements using more advanced algorithms.

To find an upper bound on performance, we used the maximum available L1 memory (i.e. a 64K SRAM and 16K cache) for the Blackfin family of embedded processors. We compare this to the L1 memory size we have already studied.

In Table 2 we show the performance benefit of the default non-optimized and fully optimized layout (i.e., KN-TR-GC) for four different L1 SRAM/Cache sizes. Five out of six of the benchmark programs achieve less less than 1% benefit for a 64K SRAM and 16K cache as compared to a 8K SRAM and 8K cache. Also, for the MPEG4 encoder benchmark program we have found that the performance of 12K SRAM and 8K cache is close to a 64K SRAM and 16K cache L1 memory.

Thus, our code layout techniques would prove beneficial in embedded systems which have low memory budgets. We show that by carefully mapping code we can achieve similar performance using a smaller L1 memory size, which will re-

Benchmark	8KS-4KC-NO-OPT	8KS-4KC-FULL-OPT	12KS-4KC-NO-OPT	12KS-4KC-FULL-OPT	8KS-8KC-NO-OPT	8KS-8KC-FULL-OPT	64KS-16KC-NO-OPT	64KS-16KC-FULL-OPT
JPEG2 encoder	1.41	1.28	1.32	1.27	1.28	1.28	1.27	1.27
JPEG2 decoder	1.39	1.12	1.34	1.11	1.15	1.11	1.10	1.10
MPEG2 encoder	11.76	10.97	11.69	10.89	10.92	10.81	10.87	10.77
MPEG2 decoder	54.41	47.27	53.25	45.92	48.03	44.82	44.37	44.33
MPEG4 encoder	72.45	47.04	70.66	44.79	49.87	44.60	43.01	42.19
MPEG4 decoder	73.31	48.22	73.31	45.43	55.68	45.31	45.29	45.05

Table 2. Table shows the executed cycles (in 10 millions) for default non-optimized and fully optimized layouts for four different L1 SRAM/Cache size.

duce the overall cost and power of the system. The demands for a better multimedia experience are always growing and applications are becoming more complex, resulting in larger code size. Our layout techniques should be extremely helpful in such cases. Our framework can also provide for possible design space exploration in L1 memory size and configurations.

7. Discussion

As a general rule of thumb, our advanced code layout techniques should be used for L1 memory configuration exploration when it is possible to only map 70-90% of the total code in L1 SRAM. If more than 90% of the code can be placed in L1 SRAM memory then the suggested techniques will not have a large impact.

To effectively capture the program characteristics, we profile the application to obtain the execution percentage of each code section, temporal reuse distances, and a function call graph. We present our analysis by treating each of these pieces of information during different algorithm steps to guide our layout. But it should be noted that the program characteristics overlap between these measurements. For example, hot code is generally frequently called code, should also have high temporal reuse. There will be few cases where a hot code has low temporal locality or a hot code is less frequently called. But as we have seen by targeting these cases and equipped with the temporal reuse measure and a function call graph we could improve performance by 3-5% on average.

During our analysis, we also noted cases where it is possible that a relatively cold function is placed in external memory and is frequently calling a hot function placed in L1 SRAM. In such cases the caller/callee functions are separated in the memory address space greater than the address range architected to make a short call. This results in an indirect call in the generated code. This introduces added instructions and requires more cycles to complete each call. If the call is frequent enough, it can degrade performance. We specifically observed this case in the MPEG2 decoder and we placed the cold caller function in L1 SRAM space (which was frequently calling a hot function already placed in L1 SRAM). Although placing the cold function in L1 SRAM displaced a relatively hot function in external memory, we observed a 2% improvement in performance for a 8K SRAM and 4K cache L1 memory size.

Optimizations based on profile guidance tend to suffer when input data sets change. Optimizations need to remain robust and work for changing program inputs. For the EEMBC consumer benchmark suite, when we ran experiments with different input data sets, we find that our results are not impacted. The execution percentages were somewhat insensitive to input changes. In general, instruction profiles are less susceptible to change due to data input changes versus data profiles.

8. Related work

Producing cache conscious code mapping which exploits the spatial and temporal locality within programs has been proposed as an effective way to reduce cache misses. Researchers over the last couple of decades have proposed different techniques for code and data remapping [18, 19, 20, 3, 4, 5, 6, 7].

Accurate program characteristics can be obtained by looking at the execution profile of a program. By carefully choosing a representative input dataset, efficient profile guided optimizations can be developed to increase program performance. Statically analyzing the program [21, 22] for compiler optimizations makes the reordering input data independent but several times it is hard to completely understand program behavior statically.

Basic block profiles and function caller/callee frequency counts have been used in code reordering algorithms. Prior work [3, 4, 5, 6, 7] has shown that procedure reordering during link time can reduce cache conflicts. Therefore, in our framework we use link time procedure reordering to effectively guide the layout but incorporating different algorithms and heuristics for the memory model we are targeting was used in [4, 5].

In [4], Hashemi et al. used a graph coloring algorithm to guide the code layout and in later work by the same group and others [6, 23] temporal reference information is considered. In our approach we use a simple graph coarsening problem instead of a graph coloring algorithm because we have the L1 SRAM memory at our disposal. Also, for the

memory model we are targeting, it is not necessary to generate a temporal reference graph. Instead, our algorithm uses a temporal reuse distance measure to partition the code sections and separate cache friendly nodes from the unfriendly ones to guide the placement in L1 SRAM .

In the embedded space, Kumar et al. [12] formulate the code and data layout problem as an integer linear programming (ILP) problem for fixed SRAM memories. But they do not address any code or data placements for effective cache layout. In [24, 25, 11], the authors present data layout techniques for embedded multimedia applications. In particular, Panda et al. [11] present a data layout algorithm for a similar configurable memory hierarchy to the systems we are targeting, but our study focuses on code layout optimizations. Also, as opposed to [11], we present our analysis for a variety of L1 memory sizes.

In [10], Tomiyama et al. formulate the code layout algorithm as an integer linear programming (ILP) problem for direct mapped caches. Our work differs from all of the above in that we address the problem of code layout for a configurable memory hierarchy system. Also, we map the code layout problem to commonly known NP-complete problems and use time-proven greedy heuristics to solve the problem efficiently. In [10], the authors have used local search algorithms to solve the ILP problem. Local search is time consuming and has been noted in their paper as one of the shortcomings of their methodology.

Researchers have proposed techniques to reduce cache conflicts with a variety of techniques as discussed above, but an on-chip L1 SRAM can be effectively used to eliminate all cache misses. In [8], we showed the benefits of using on-chip L1 SRAM for the MPEG-2 encoder benchmark program for both code and data management. The code layout algorithm discussed in our previous work [8] used cache misses as a metric to estimate the cycles lost and guide the mapping. Given that we have caller/callee frequency counts and reuse distances, we do not have to infer the relationship between different functions. Also, in [8] we evaluated function overlays for managing code sections. Our evaluation of our overlay manager was hand-tuned. But by solving the graph coarsening problem, the resulting merged nodes will be automatic candidates for implementation with function overlays.

9. Conclusions

In this paper, we presented a code layout framework for configurable code memory systems now typical in embedded processors. Our tool allows for automatic code layout and extensive design space exploration. This should reduce the time to market embedded products. We presented performance improvements for industry standard multimedia benchmark programs, evaluating them on real hardware using performance counters. We show performance benefits ranging from 6% to a maximum of 35%, when using the default linker layout as a baseline. We achieve a 19% improvement on average for the 6 benchmark programs. Our future work will focus on developing algorithms for data layout and considering tradeoffs in code and data layout in shared memory spaces.

References

[1] Analog Devices Incorporation. Blackfin processor hardware reference manual. Norwood, MA, USA, 2003.

[2] LSI Logic Corporation. Cw33000 risc microprocessor core hardware manual. 1992.

[3] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, New York, NY, USA, 1990. ACM Press.

[4] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 1997. ACM Press.

[5] Nicholas C. Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. Procedure placement using temporal ordering information. In *MICRO*, pages 303–313, 1997.

[6] John Kalamatianos, Alireza Khalafi, David R. Kaeli, and Waleed Meleis. Analysis of temporal-based program behavior for improved instruction cache performance. *IEEE Trans. Comput.*, 48(2):168–175, 1999.

[7] John Kalamatianos and David Kaeli. Accurate simulation and evaluation of code reordering. In *Proceedings of the IEEE International Symposium on the Performance Analysis of Systems and Software*, pages 45–54, April 2000.

[8] Kaushal Sanghai, David Kaeli, and Richard Gentile. Code and data partitioning on the blackfin 561 dual-core platform. In *ODES-3: Digest of the third workshop on Optimizations for DSP and Embedded Systems*, pages 92–100, 2005.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[10] Hiroyuki Tomiyama and Hiroto Yasuura. Optimal code placement of embedded software for instruction caches. In *EDTC '96: Proceedings of the 1996 European conference on Design and Test*, page 96, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):682–704, 2000.

[12] T.S. Rajesh Kumar, R. Govindarajan, and C. P. Ravi Kumar. Optimal code and data layout in embedded systems. In *16th International Conference on VLSI Design*, pages 573–579, 2003.

[13] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Cache-aware scratchpad allocation algorithm. In *DATE '04:*

Proceedings of the conference on Design, automation and test in Europe, page 21264, Washington, DC, USA, 2004. IEEE Computer Society.

[14] EEMBC. Embedded microprocessor benchmark consortium.

[15] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[16] Thierry Lafage and Andre Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. pages 145–163, 2001.

[17] Amir Hashemi. Efficient program mapping for improved cache performance. *Ph.D. thesis,Northeastern University, Boston, MA*, 1997.

[18] S. McFarling. Program optimization for instruction caches. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 183–191, New York, NY, USA, 1989. ACM Press.

[19] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34(5):1–12, 1999.

[20] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 140–153, New York, NY, USA, 2002. ACM Press.

[21] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[22] Abraham Mendlson, Shlomit S. Pinter, and Ruth Shtokhamer. Compile time instruction cache optimizations. *SIGARCH Comput. Archit. News*, 22(1):44–51, 1994.

[23] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.

[24] Chidamber Kulkarni, Francky Catthoor, and Hugo De Man. Advanced data layout optimization for multimedia applications. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 186–193, London, UK, 2000. Springer-Verlag.

[25] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor, and H. de Man. Cache conscious data layout organization for embedded multimedia applications. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 686–693, Piscataway, NJ, USA, 2001. IEEE Press.

[26] Sundaram Anantharaman and Santosh Pande. An efficient data partitioning method for limited memory embedded systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 108–222, London, UK, 1998. Springer-Verlag.

[27] Christophe Guillon, Fabrice Rastello, Thierry Bidault, and Florent Bouchez. Procedure placement using temporal-ordering information: dealing with code size expansion. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 268–279, New York, NY, USA, 2004. ACM Press.

[28] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *SIGPLAN Not.*, 29(11):252–262, 1994.

[29] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*, pages 104–109, Washington, DC, USA, 2004. IEEE Computer Society.

Keynote:

Challenges with Concurrency and Time in Embedded Software

Edward A. Lee
University of California at Berkeley

Register Allocation for Processors with Dynamically Reconfigurable Register Banks

Ralf Dreesen, Michael Hußmann, Michael Thies and Uwe Kastens
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany
{rdreesen,michaelh,mthies,uwe}@uni-paderborn.de

March 11, 2007

Abstract

This paper presents a method for optimizing register allocation for processors which can reconfigure their access to different register banks to increase the number of accessible registers.

Register allocation like the graph coloring method of Chaitin aim at keeping a large number of values in registers and minimize the amount of necessary spill code.

We extend that method in two directions: First, reconfiguration code is needed to switch between registers banks. Those code costs, too, are minimized. Second, our method chooses the values to live together in a register bank, such that reconfiguration costs are reduced.

For the latter task, a novel program analysis is proposed, which applies a DFA to yield information on register access patterns.

The register allocation method can be extended for a multi-core processor with a shared reconfigurable register bank.

1 Introduction

This paper proposes a method, which performs a *register allocation* for a reconfigurable register bank. The proposed method inserts *reconfiguration instructions* into the program code, to make different sets of registers accessible. It is also suitable for processors with multiple synchronous cores, as will be shown at the end of this paper.

To discuss the register allocation method, the properties of the reconfigurable register bank are described first.

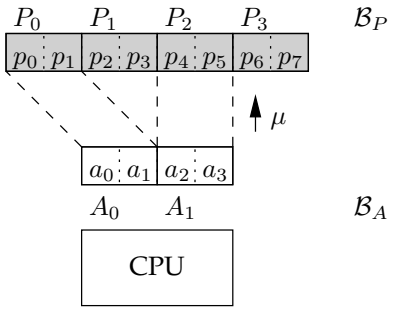

Figure 1: Schematic reconfigurable register bank

Reconfigurable Register Bank The reconfigurable register bank consists of a large set of registers. The registers can not be accessed directly, but only through a smaller set of register frames, which are actually encoded as operands in instructions. In the following, the registers which actually store values are called *physical registers* and register frames are called *architectural registers*.

Reconfiguration The mapping of physical registers into architectural registers is changed explicitly at runtime by *reconfiguration instructions*, which are part of the executed program.

To reduce the number of reconfiguration instructions, each change of a mapping affects a block of registers, instead of just one register. Therefore, the physical and architectural registers are divided into equal sized blocks of registers. Hence, a reconfiguration instruction maps a *physical register block* into an *architectural register block*.

Terminology The schema of a reconfigurable register bank is shown in figure 1. Physical registers are

denoted by p_i and architectural registers by a_i. The symbols P_i refer to physical blocks and A_i to architectural blocks. The set of all physical blocks is given by $\mathcal{B}_P$ and the set of architectural blocks by $\mathcal{B}_A$.

The mapping of physical blocks into architectural blocks is given by the function μ, which maps an architectural block to a physical block. In figure 1 μ maps A_0 to P_0 and A_1 to P_2. If the mapping of an architectural block can not be predicted at compile time, it is said to be mapped to bottom ($\perp$).

Overview The proposed register allocation method extends the graph coloring approach of Chaitin [1]. Therefore, affinities between virtual registers v_i and physical blocks P_j are computed (section 3.1). Affinities estimate the number of reconfiguration instructions that are avoided by assigning v_i to a physical register of P_j. The affinities are used during coloring as a secondary criterion for choosing a physical register.

Reconfiguration instructions are inserted into the program using the "as late as possible" strategy, i.e. right before an access to an unmapped physical register (section 3.2). An architectural block for the mapping is chosen according to the replacement strategy "Latest Used Again". In order to reuse mappings that are established in preceding basic blocks, this paper presents a method to define mapping conventions between basic blocks.

We discuss our register allocation method for a single processor first. Section 3.3 describes minor modifications needed for multi-core processors.

2 Related Work

Our register allocation method builds on classic graph coloring as introduced by Chaitin [1]. Chaitin reduces the problem of assigning virtual registers to physical registers to a graph coloring problem. We exploit some freedom that is left in this algorithm, to model reconfiguration of register banks and to minimize reconfiguration code. Reconfiguration follows the replacement strategy proposed by Belady [2]. Originally a paging strategy for MMUs, it is optimal with respect to the number of page replacements. It can also be applied as a register allocation strategy for straight-line code. Driven by limited information on future register accesses, we use Belady's strategy to choose the best physical block to be replaced. This reduces the number of reconfiguration instructions.

In the following, we describe two restricted forms of register reconfiguration, which are used in widespread conventional processors. After that, we outline two scientific reconfigurable processor designs and relate their associated register allocation methodologies to our approach.

Register Renaming Register renaming [3] is a hardware mechanism to avoid register access constraints caused by pending memory accesses. If the content of a register r is to be stored, it must not be overwritten by a succeeding instruction, until the store is completed. Register renaming circumvents this restriction by binding a different physical register to r. This way, the pending store instruction can access the old value of r, while the new value is stored in another physical register.

In contrast to our approach, register renaming does not increase the number of compiler-exploitable registers, but is limited to shortening register life times by a fixed amount.

Register Windowing Register windowing, as described in [4], is a mechanism to make different sets of registers accessible using a sliding window. Its primary purpose is to avoid memory accesses caused by register save and restore code on function calls. Register windowing constitutes a special case of our model: Only adjacent physical blocks can be mapped into architectural blocks. Moreover, repeated `save` and `restore` instructions might be necessary, to make a physical block accessible. Thereby, it is not possible to select a *specific* physical block for replacement. As a result of these restrictions, register windowing is not viable for code with interspersed accesses to many registers.

Register Connection Kiyohara et al [5] propose a family of register architectures with two classes of registers, namely the *core registers* and the *extended registers*. With their most elaborate architecture, core registers can be accessed without additional effort, whereas every access to an extended register must be preceded by a reconfiguration instruction. The register allocation method assigns heavily accessed virtual registers to core registers, whereas the remaining registers are assigned to extended registers. In contrast to our register architecture, registers are not organized in blocks and thus an affinity analysis is not necessary. If the number of core registers is exceeded, many reconfiguration instructions are likely to be inserted to prepare access to extended registers.

Windowed Register File The register architecture as proposed by Ravindran et al [6] divides physical registers into banks, where only one bank at a time can be accessed. As a result, additional copy instructions (inter-window-moves) are necessary to allow simultaneous access to registers in different banks.

To estimate reconfiguration and copy costs, affinities are used to partition the registers into banks.

The affinities estimate the reconfiguration costs only roughly. Using a special version of our analysis, the reconfiguration costs could even be determined exactly.

The architecture of Ravindran et al lends itself better to an exact estimation of reconfiguration costs, but incurs the aforementioned inter-window-move instructions.

3 Register Allocation

In the register allocation phase of the compiler, the virtual registers as used in the abstract machine program are assigned to the physical registers of the processor. With a reconfigurable register bank, the same task is to be solved. Hence, the virtual registers are *assigned* to the physical registers in a *first phase*. In addition, *reconfiguration instructions* need to be inserted in a *second phase*, to make the physical registers accessible to the processor via the architectural registers. See Fig. 6 for an example of the resulting code.

3.1 Allocation of Physical Registers

This section describes the register allocation algorithm that assigns the virtual registers to the physical registers. The allocation algorithm augments the conventional register allocation of Chaitin [1] which is based on graph coloring. In addition, the algorithm aims to reduce the number of reconfiguration instructions during the second phase. Therefore, an estimate of anticipated reconfiguration costs is used, to partition the virtual registers into the physical blocks.

Idea of the Heuristic The basic idea of the heuristic is to assign virtual registers that are often accessed in conjunction to physical registers of the same physical block. Hence, if two virtual registers are accessed close together, and both are assigned to physical registers of the same physical block, only one reconfiguration instruction is needed to make both virtual registers accessible.

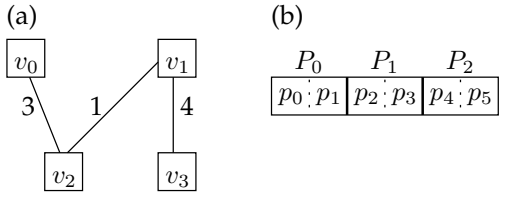

Figure 2: Example of an affinity graph.

Affinity Graph The decision, which virtual registers are assigned to the same physical block, is based on the number of reconfiguration instructions that are avoided by this grouping of virtual registers. The number of reconfiguration instructions, that are avoided by assigning two virtual registers to the same physical block, is expressed by the *affinity* between both virtual registers. The affinities between all pairs of virtual registers are represented in a so-called *affinity graph*. The nodes of this graph correspond to the virtual registers and each edge is weighted with the affinity between two virtual registers. The affinities are derived from the access patterns of the virtual registers. A detailed description is given below.

Figure 2a shows an example of an affinity graph for the virtual registers $v_0, \ldots, v_3$. The Figure 2b aside shows a physical register bank with 6 physical registers that are divided into 3 physical blocks. If the virtual register v_0 is assigned to the physical register p_0 and v_2 is assigned to p_1, 3 reconfiguration instructions are avoided, since the affinity between v_0, v_2 is 3 and they are assigned to the common physical block P_0.

During the allocation process, some of the virtual registers are already assigned to physical registers and thereby to physical blocks. The virtual register nodes in the affinity graph are therefore replaced by nodes representing physical blocks and the affinities are recomputed. In this way, affinities between virtual registers and physical blocks emerge that represent the number of avoided reconfiguration instructions, if the virtual register is assigned to the physical block.

So far, the interpretation of the affinity has been described. In order to *define* the affinity, the replacement strategy for register blocks needs to be discussed.

Replacement Strategy If a physical block must be made accessible at a program position s, the compiler needs to decide, which architectural block should host the physical block. The first choice is,

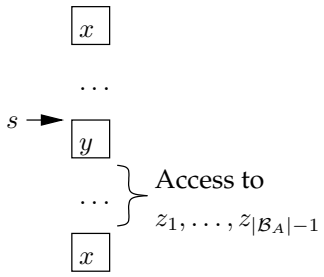

Figure 3: Replacement of physical block between two accesses.

to use an unmapped architectural block. If no such block exists, an architectural block A must be chosen that is already mapped to some physical block P as given by $\mu(A)$. Which block A is chosen depends on its current content P and is determined by the *replacement strategy.*

Belady The compiler uses the replacement strategy of Belady [2] that replaces the mapping of the architectural block A, whose physical block $P = \mu(A)$ is *latest used again.* In the following, this strategy is referred to as LUA. The strategy of Belady [2] also defines that a physical block is made accessible as late as possible (ALAP), i.e. right before the next access to the physical block.

Originally, this strategy has been presented by Belady as paging algorithm for virtual memory management and is also used for local register allocation in basic blocks.

Persistence of Physical Blocks For the affinity analyses it is necessary to determine, if a physical block might be replaced between two accesses to the physical block. The following theorem expresses, in which case a physical block is definitely *not* replaced between two accesses.

Theorem 1 *According to LUA, a physical block x is not replaced between two accesses to x, if less than $|\mathcal{B}_A|$ pairwise different blocks are accessed in between.*

Proof 1 *This theorem is proven by contradiction. Let x be replaced by y at a position s between two accesses to x, as shown in Fig. 3. Then all architectural blocks must be mapped at position s. Let the other architectural blocks further be mapped to $z_1, \ldots, z_{|\mathcal{B}_A|-1}$. Since x is replaced, the physical blocks $z_1, \ldots, z_{|\mathcal{B}_A|-1}$ must be accessed between s and the second access to x. All in all, with $z_1, \ldots, z_{|\mathcal{B}_A|-1}$ and y, $|\mathcal{B}_A|$ physical blocks need to be accessed between the two accesses to x, in order to replace x.*

3.1.1 Definition of Affinity

As already mentioned, the affinity between two virtual registers estimates the number of reconfiguration instructions that are avoided, if both virtual registers are assigned to the same physical block. This section gives an equivalent definition of the affinity that is used afterwards to compute affinities efficiently. The definition is based on so-called *access pairs* for two virtual registers. In the following, the symbol v_i denotes an access to a virtual register v.

Definition 1 (Affinity of virtual registers) *Let x and y be two virtual registers. The affinity $\alpha(x, y)$ between x and y is defined as the number of unordered access pairs (x_i, y_j) which have the following properties:*

1. *The number of pair-wise different virtual registers that are accessed between x_i and y_j is less than $|\mathcal{B}_A|$.*
2. *x and y are not accessed between x_i and y_j.*

Let x and y be assigned to the same physical block P and let (x_i, y_j) be an access pair. Obviously, less than $|\mathcal{B}_A|$ other physical blocks are accessed between x_i and y_j, because less than $|\mathcal{B}_A|$ virtual registers are accessed by definition. Consequently, P is not replaced between x_i and y_j according to Theorem 1, i.e. no further reconfiguration is needed to access y_j. Otherwise, it *might* be necessary to insert a reconfiguration instruction between x_i and y_j. Thus, a reconfiguration instruction is avoided for this access pair, if x and y are assigned to the same physical block.

The second condition in Def. 1 ensures that each avoided reconfiguration instruction is counted only once.

3.1.2 Construction of Affinity Graphs

In order to calculate the complete affinity graph, all access pairs are determined for each pair of virtual registers. This task can be solved for all virtual registers, by regarding for each access v_i, the $|\mathcal{B}_A|$ different virtual registers that are accessed after v_i.

Figure 4a shows an example for $|\mathcal{B}_A| = 2$ and $v_i = x_1$, in which the access pairs (x_1, a_1) and (x_1, b_1)

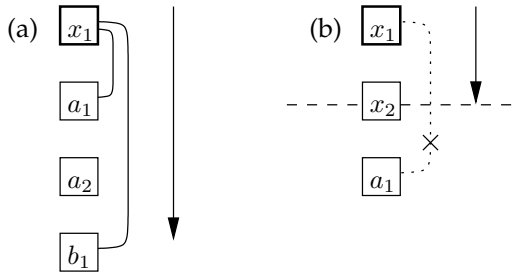

Figure 4: Calculation of affinity.

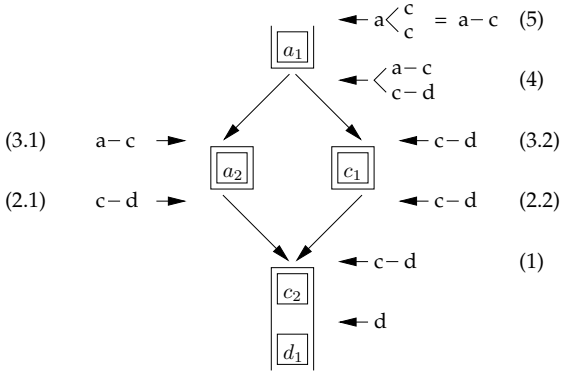

Figure 5: Example of n-liveness analysis.

are found, by regarding x_1 and the following 2 register accesses. The repeated access a_2 is ignored in order to ensure the second condition of Def. 1. For the same reason, all register accesses following the repeated access x_2 are ignored, as indicated in Fig. 4b.

To find all access pairs for a regarded register access v_i, the $|\mathcal{B}_A|$ virtual registers that are accessed next need to be known. For $v_i = x_1$, this would be (a, b) in example 4a and (x, a) in example 4b. These registers can be computed by our n-liveness analysis that is outlined in the following section.

3.1.3 Analysis of n-Liveness

The n-liveness analysis is a static program analysis that applies data flow analysis to determine, for each program position, the n distinct registers that are accessed next. As already mentioned, it is used for the computation of the affinity graph. In addition, its results are used for the placement of reconfiguration instructions, as described in section 3.2.

Trees Figure 5 shows an exemplary control flow graph that is annotated with the results of the n-liveness analysis for each program position.

The results are given as trees of n-live registers. In the figure, the trees are drawn from left to right, i.e. the root is on the left side. The depth of the trees is limited by n, that is $|\mathcal{B}_A| = 2$ for the example in Fig. 5.

Position (1) is annotated with the degenerated tree `c-d`, meaning that `c` and `d` are accessed next in this very order. The same holds of course for the end of both preceding basic blocks at position (2.1) and (2.2). If virtual registers are accessed repeatedly, only the first access is considered as can be seen at position (3.2) where the access c_2 is ignored. At position (4), the register access sequence can not be predicted statically due to the splitting of the control flow. Therefore, the tree reflects all possible access sequences that might occur at runtime. The left access sequence contributes the tree branch `a-c`, whereas the right sequence adds the branch `c-d`. If a node has common sub-nodes, the tree is simplified like at position (5), without loosing any relevant information.

Tree Size Due to the limited depth and degree of each tree, the number of nodes is limited. Although the tree can theoretically consist of many nodes, it is usually rather small as basic blocks typically contain accesses to multiple virtual registers. This results in linear chains without any branches in the tree. Furthermore, the degree of a branch in the tree is limited by the degree of the corresponding control flow node. Most control flow nodes have no more than two successors.

Computation The n-live registers are computed using the general method of an iterative data flow analysis. This means that the trees of n-live registers are computed iteratively for the end and begin of each basic block. The convergence of the n-liveness analysis is not immediately obvious, because it can not be formulated as a DFA problem, which operates on a lattice. Nevertheless, we have proven the convergence of the n-liveness analysis in [7].

Probabilities To supply quantitative information about the likelihood of register accesses occurring at runtime, the nodes in these trees are annotated with probabilities. These probabilities are derived from the branching probabilities in the control flow graph. The probabilities are used as weights in several heuristics. In addition, probabilities allow us, to recognize accesses as certain, even after a divergence in control flow.

To summarize, the n-liveness analysis determines, for each program position, if a virtual register v belongs to the n virtual registers that are accessed next. This knowledge can in turn be used, to calculate the affinity graph. With the help of the affinity graph, the virtual registers are assigned to the physical registers, such that virtual registers with high affinity are assigned to the same physical block. Thus, most physical register accesses will not require a reconfiguration instruction, since the corresponding physical block is already accessible.

3.2 Reconfiguration

The register allocation described so far assigns the virtual registers to the physical registers of the processor. In the second phase, the reconfiguration instructions are inserted. The inserted reconfiguration instructions implicitly determine the mapping from physical to architectural registers for each program position. Hence, the physical register operands can be rewritten to refer to architectural registers.

At first, the insertion of reconfiguration instructions will be shown for a single basic block. Afterwards, this method is extended to support mappings across basic blocks.

3.2.1 Intra-Block Reconfiguration

The basic idea of the algorithm is to keep track of the mapping function μ, while iterating over the instructions of a basic block.

At the begin of a basic block, the mapping function μ is pessimistically assumed to be undefined for all architectural blocks, i.e. $\forall A : \mu(A) = \perp$ holds.

If an instruction i is found, that accesses a register of an inaccessible physical block P, a reconfiguration instruction is inserted directly before i to make P accessible. The reconfiguration instruction establishes a mapping $\mu(A) = P$, where A was either unmapped or mapped to the physical block P' that is latest used again.

Example Figure 6 shows an example of code in a basic block on the right with accesses to the physical blocks P_0, P_1, P_2. On the left, the current mapping function μ is indicated for a register architecture with two architectural blocks A_0, A_1. Since the architectural blocks are assumed to be unmapped at the begin, a reconfiguration instruction $\mu(A_0) := P_0$ needs to be inserted at the very begin of the basic block, to make P_0 accessible. The same applies for the second reconfiguration instruction prior to the access to

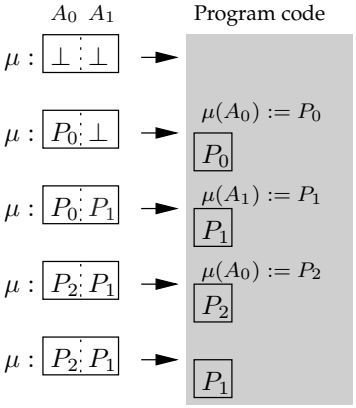

Figure 6: Insertion of reconfiguration instructions according to LUA.

P_1. The third reconfiguration instruction needs to replace a mapping, since both architectural blocks are already mapped to physical blocks. According to the LUA strategy the mapping $\mu(A_0) = P_0$ is replaced, as P_0 is latest used again.

3.2.2 Inter-Block Reconfiguration

Up to now, the mappings at the begin of a basic block have been assumed to be undefined. For this reason, reconfiguration instructions had to be inserted at the begin of *every* basic block.

IN and OUT Mappings In order to reuse the mappings from preceding basic blocks, the mappings at the begin and end of a basic block are decided in a preceding step. The mapping function for the begin of a basic block is denoted μ_{IN} and specifies, which mappings can be assumed at the begin of the basic block. Accordingly, the mapping function μ_{OUT} specifies the mappings that *must* be assured at the end of a basic block.

$\mu_{OUT} \rightarrow \mu_{IN}$ **Dependency** To assume $\mu_{IN}(A) = P$ at the begin of a basic block b, $\mu_{OUT}(A) = P$ must hold in all preceding basic blocks $pred(b)$. If A is mapped to different physical blocks in μ_{OUT} of the predecessors, the mapping of A is undefined ($\perp$) in the IN mapping. This can be expressed formally as:

$$\mu_{IN\,b} = \bigcap_{\hat{b} \in pred(b)} \mu_{OUT\,\hat{b}}$$

where $\cap$ is defined as

$$\mu_x(A) \cap \mu_y(A) = \begin{cases} P & \text{for } \mu_x(A) = \mu_y(A) = P \\ \perp & \text{else} \end{cases}$$

Selection of Physical Blocks The definition of μ_{IN} aims to reduce the number of reconfiguration instructions at the begin of a basic block b. Therefore, if P is n-live at the begin of b, a mapping $\mu_{IN}(A) = P$ is arranged, if possible. The n-liveness guarantees that P persists from the begin of the basic block up to the first access to P. Thus, a reconfiguration instruction is definitely avoided in b.

Effect on Predecessors To guarantee the mapping $\mu_{IN}(A) = P$, the same mapping must be defined in μ_{OUT} of all preceding basic blocks. Unfortunately, a mapping $\mu_{OUT}(A) = P$ can introduce an additional reconfiguration instruction in the predecessor. To predict, whether a reconfiguration instruction must be added to ensure $\mu_{OUT}(A) = P$, the maximum cut between the last access to P and the end of the basic block is regarded. If the maximum cut is smaller than $|\mathcal{B}_A|$, the physical block P will persists until the end of the preceding basic block according to LUA. Hence, this case requires no additional reconfiguration instruction in the predecessor.

Estimation of Costs The disadvantage of an additional reconfiguration instruction in a predecessor is expressed in terms of *costs*. The costs of a reconfiguration instruction in a block b is given by the execution frequency of the basic block. Hence, the costs for a mapping $\mu_{IN}(A) = P$ is given by the sum of costs of additional reconfiguration instructions in the preceding basic blocks. If these costs are lower than the costs of a reconfiguration in the basic block, the mapping will be defined. As a result, reconfiguration instructions are moved out of loops into a block preceding the loop.

3.3 Extensions for Multi-Core Processors

So far, the register allocation has been discussed for a single-processor. This section outlines how the register allocation can be extended for processors with multiple synchronous cores, reconsidering only few aspects. The processor is assumed to have the structure shown in Fig. 7.

It consists of a shared physical register file that can be accessed uniformly by every core. Each core has its own set of architectural registers to establish its mapping independently.

Since the cores access a shared register file, the creation of the conflict graph and the coloring treat parallel instructions like a single VLIW instruction,

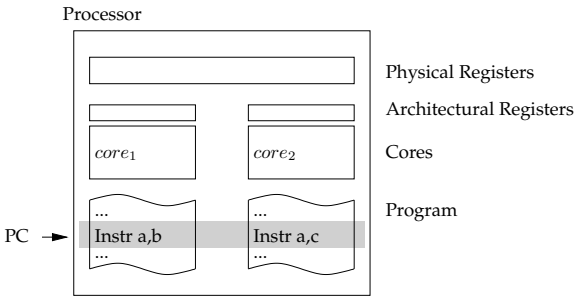

Figure 7: Multi-core processor architecture.

whereby the conventional algorithms for single processors can be applied.

The affinity analysis and the insertion of reconfiguration instructions are performed for each core separately, since each core has a dedicated architectural register bank.

4 Evaluation

This section compares the efficiency of a reconfigurable register bank to static register architectures, using several benchmark programs.

First, the tool chain of the evaluation and the benchmarks is outlined. Afterwards, several aspects of the reconfigurable register architecture are analyzed.

Tool Chain The register allocation algorithm has been implemented in an optimizing compiler for a streamlined RISC processor called S-Core. The processor is described in [8] and was developed in the GigaNetIC project. The compiler augments the lcc [9] frontend with a complete suite of global optimizations and conducts global register allocation [1]. For details about the compiler, refer to [10].

The benchmarks were evaluated using a cycle accurate simulator. Most instructions take 1 cycle on the S-Core RISC processor. Memory read and write instructions have been assumed to take 3 cycles on average (cache hit: 2 cycles; cache miss: 5 cycles) Reconfiguration instructions have been counted as 2 cycles. A recent virtual hardware prototype has shown that reconfiguration should be feasible even in one cycle.

Benchmarks In order to utilize the 32 registers of the reconfigurable register bank, benchmarks with an inherent high register pressure have been chosen. Nevertheless, high register pressure also results

from preceding compiler optimizations, like "common subexpression elimination", "register coalescing", and certain scheduling strategies, like "software pipelining".

For each benchmark, the maximum cut of the virtual register's life ranges is shown in Table 1. The maximum cut is a lower bound for the number of registers that are required to avoid all spill code.

In the following, a short description of each benchmark is given

convolution computes the discrete convolution of a 100 element array with a 32 element array. The results are written into another 100 element array.

fftlike simulates the variable access pattern of a fast fourier transformation. The intermediate results are alternately stored in two arrays of 16 elements.

median computes the median of each n consecutive elements in an array of 100 elements. The computation is executed simultaneously for $n = 1, 2, 3, \ldots, 32$ in a single pass.

mm4 multiplies two 4×4 matrices. The result is again stored in a 4×4 matrix

poly evaluates a polynomial of degree 32 with variable coefficients. The evaluation is repeated 100 times.

Register Architecture Figure 8 shows the number of simulation cycles for a normal and a reconfigurable register architecture. In the case of the normal register architecture, the processor accesses the registers directly. For the reconfigurable register architecture, the processor accesses 32 physical registers via 16 architectural registers. Thus, the instruction set is the same in both cases. Furthermore, the architectural registers are divided into 4 blocks of 4 registers each in case of the reconfigurable architecture.

Since the benchmarks have a high register pressure, they benefit from the additional 16 registers that are offered by the reconfigurable architecture. Table 1 lists the exact number of execution cycles and the number of inserted reconfiguration instructions. It is remarkable that the number of simulation cycles decreases by about 30% percent for the reconfigurable architecture.

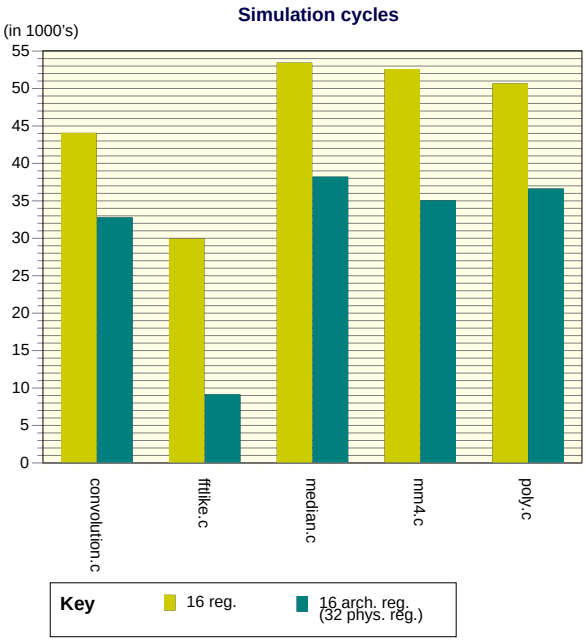

Figure 8: Simulation results for different register architectures.

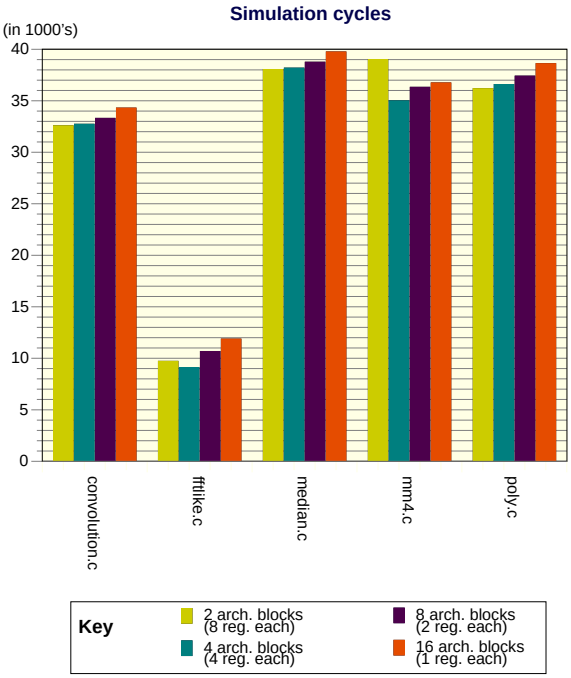

Figure 9: Simulation results for different block sizes.

Block Size In the previous benchmark, the architectural registers have been divided into 4 blocks. In this section, the influence of the number of blocks is analyzed for a fixed number of 16 architectural and

Benchmark	Maxcut	Cycles (16 regs.)	Cycles (32 reconf. regs.)	Reconf. instr.	Total instr.
convolution	32	44078	32786	22	260
fftlike	35	29958	9169	36	205
median	33	53450	38232	13	329
mm4	44	52582	35054	34	285
poly	22	50660	36646	9	83

Table 1: Benchmark characteristics and results.

32 physical registers.

For the benchmarks `poly`, `convolution` and `median`, the number of simulation cycles increases with a growing number of architectural blocks, compare Fig. 9. This behavior is caused by the access pattern of the benchmarks. The registers are accessed repeatedly in the same order. Hence, at least $|\mathcal{B}_A|$ reconfiguration instructions must be inserted, to switch between the 32 physical registers.

In contrast to that, the matrix multiplication benchmark accesses rows and columns of 4 registers repeatedly. Thus, a block size of 4 architectural registers performs best for this benchmark.

For larger programs smaller block sizes require more reconfiguration instructions to establish well-defined mappings, for example around each function call. Also, smaller block sizes limit the effect of each reconfiguration instruction to fewer register accesses. Kiyohara [5] constitutes an extreme case with one reconfiguration instruction per access to an extended register and a fixed set of core registers, which can be accessed without any reconfiguration.

Our approach provides a similar distinction, based on the access frequency of a register. If a register like the stack pointer is accessed frequently, its block will not be unmapped by our register allocation.

Affinity As mentioned in section 3.1, the affinity analysis intends, to reduce the number of reconfiguration instructions.

Figure 10 shows the actual effect of the affinity analysis, compared to a random assignment of virtual registers to physical register blocks. The number of simulation cycles decreases by up to 50% due to reduced number of reconfiguration instructions.

Code Density For our benchmark, reconfiguration instructions increase code size by about 12% on average. This result is based on an architecture with 16 architectural and 32 physical registers. Instructions are encoded in 16 bits and support at most two register operands. Thus, an ideal extension of this

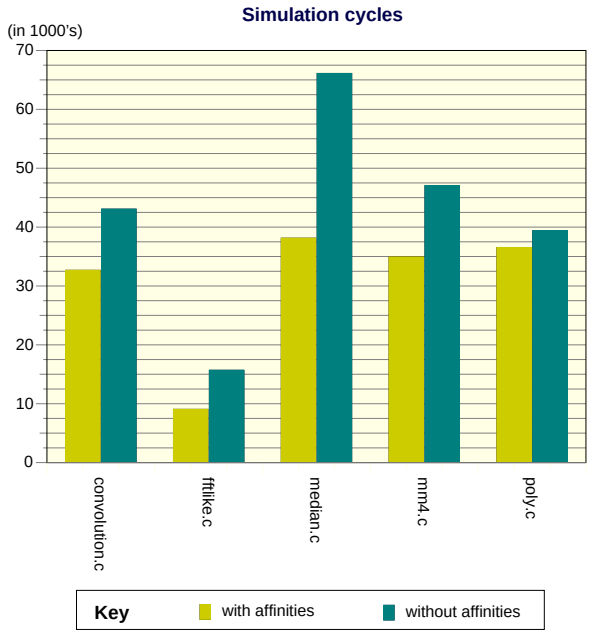

Figure 10: Effects of the affinity analysis.

architecture to 32 freely addressable registers would require instructions of 18 bits minimum. This results in code growth of 12.5% or more.

Realistically, the instruction format would have to be extended to 24 or even 32 bits, which cuts code density in half and leaves a huge share of the opcode space unused. In contrast, reconfigurable architectures constitute a more flexible means to add more registers, while preserving the overall instruction format.

5 Conclusion

A reconfigurable register architecture offers additional registers to the executing program, at additional cost for reconfiguration code. The instruction format remains unchanged. Programs with high inherent register pressure, and compiler optimizations,

which increase register pressure, do benefit from these additional registers.

We have described an augmented register allocation phase for a compiler, which utilizes all physical registers and manages their mapping to architectural registers automatically. Static program analyses are used to exploit the block structure of the reconfigurable register bank and to reduce reconfiguration overhead at runtime.

The evaluation of the register block size has shown, that reasonably sized register blocks yield a modest reconfiguration overhead. Large register blocks suit especially repeated accesses to many registers. For programs with irregular access sequences, smaller block sizes yield slightly better results.

Our affinity analysis further reduces the number of reconfiguration instructions. For programs with high register need, offering two physical registers per architectural register (instead of one) yields a speedup of about 30%.

References

[1] Chaitin, G.J.: Register allocation & spilling via graph coloring. In: SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, New York, NY, USA, ACM Press (1982) 98–101

[2] Belady, L.A.: A study of replacement algorithms for virtual-storage computer. IBM Systems Journal **5**(2) (1966) 78–101

[3] Tomasulo, R.: An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. **11**(1) (1967) 25–33

[4] SPARC International: The SPARC architecture manual: Version 8. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1992)

[5] Kiyohara, T., Mahlke, S., Chen, W., Bringmann, R., Hank, R., Anik, S., Hwu, W.M.: Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. In: ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture, New York, NY, USA, ACM Press (1993) 247–256

[6] Ravindran, R.A., Senger, R.M., Marsman, E.D., Dasika, G.S., Guthaus, M.R., Mahlke, S.A., Brown, R.B.: Increasing the number of effective registers in a low-power processor using a windowed register file. In: CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, New York, NY, USA, ACM Press (2003) 125–136

[7] Dreesen, R.: Registerzuteilung für Prozessor-Cluster mit dynamisch rekonfigurierbaren Registerbänken. Diploma-Thesis, University of Paderborn (2006)

[8] Langen, D., Niemann, J.C., Porrmann, M., Kalte, H., Rückert, U.: Implementation of a RISC Processor Core for SoC Designs - FPGA Prototype vs. ASIC Implementation. In: Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC), Hamburg, Germany (2002)

[9] Fraser, C.W., Hanson, D.R.: A Retargetable C Compiler: Design and Implementation. Benjamin/Cummings Pub. Co., Redwood City, CA, USA (1995)

[10] Bonorden, O., Brüls, N., Le, D.K., Kastens, U., Meyer auf der Heide, F., Niemann, J.C., Porrmann, M., Rückert, U., Slowik, A., Thies, M.: A holistic methodology for network processor design. In: Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003). (2003) 583–592

Compiler Optimization for the Cell Architecture

Junggyu Park, Alexander Kirnasov, Daehyun Cho, Byungchang Cha, Hyojung Song
Samsung Electronics
{junggyu.park, a78.kirnasov, dae.hyun.cho, bc.cha, hjsong}@samsung.com

Abstract

Multiprocssor-System-on-a-Chips(MPSoCs) are being increasingly employed to solve a diverse spectrum of problems from both high-end and low-end computing. Among them, the Cell Broadband Engine (CBE) processor is for high-end computing, targeting such application as multimedia streaming, game applications as in PlayStation 3, and other numerically intensive workloads. Developing optimized software for the CBE, however, is very difficult because of its complex architecture and multilevel heterogeneous parallelism. In this paper, we present an overview of a GCC-based compiler projects for the CBE, and show implementation of optimizations that we developed for the SPE processor of the CBE. Our goal in this project has been to provide both high performance and programmability, though in this paper we focus on performance. We evaluated the performance of our compiler using several well-known benchmark applications and a H.264 codec running on a CBE processor. Results indicate that significant speedup can be achieved including 21% improvements from H.264 codec.

1. Introduction

Multiprocessor-System-on-a-Chips (MPSoCs) have been widely used in today's high performance embedded systems, such as network processors, mobile phones, and MP3 players. OMAP from Texas Instruments, Nexperia from Philips, MDSP from Cradle[6], and Cell Broadband Engine (CBE) from Sony/Toshiba/IBM[5] show how MPSoCs can be used successfully in the mobile, portable, or CE devices. Among them, the CBE processor is for high-end computing, targeting such application as multimedia streaming, game applications as in PlayStation 3, and other numerically intensive workloads. It can provide both performance and flexibility to such devices as digital TV by allowing software codec to run as fast as hardware codec. Developing optimized software for the CBE, however, is very difficult because of its complex architecture and multilevel heterogeneous parallelism - it is an asymmetric multiprocessor system with one PowerPC core and eight synergistic processor elements (SPEs) for computation intensive workloads, and each SPE consists of a SIMD processor having two pipelines designed for streaming workloads, a local memory, and a globally coherent DMA engine.

Many attempts have been tried for the easy development of fast software against the complexity of the CBE. Toshiba has introduced media framework which addressed thread level parallelism [3]. But optimizations by instruction level parallelism (ILP) and data parallelism should be done by optimizing compiler or manual vectorization. RapidMind provides automatic parallelism and ILP, but it also needs manual vectorization to get more performance [7]. IBM introduced compiler techniques to address all of the problems above [9]. What we are trying to do is similar to the last one, but focused more on multimedia workloads, especially codec.

In this paper, we present compiler optimizations that we developed for SPE processor, together with the overview of our GCC-based compiler project for the CBE. Our goal in this project has been to provide both high performance and programmability, though our main contribution in this paper is a set of compiler techniques that provide high levels of performance for the SPE processors. Because of large number of register files, we do not care much about register pressure in swing modulo scheduling (SMS, a software pipelining approach), but instead care about blocking situation that often makes SMS fail. Aggressive if-conversion is performed to reduce big penalty of branch, even integrated with SMS. SPE specific features such as memory alignment, 2 pipelines, branch hint, 64-byte unit of instruction fetch are also considered. For data and thread parallelism, automatic vectorization and OpenMP supports are under development. Since our first priority customer is streaming applications (especially codec), we used

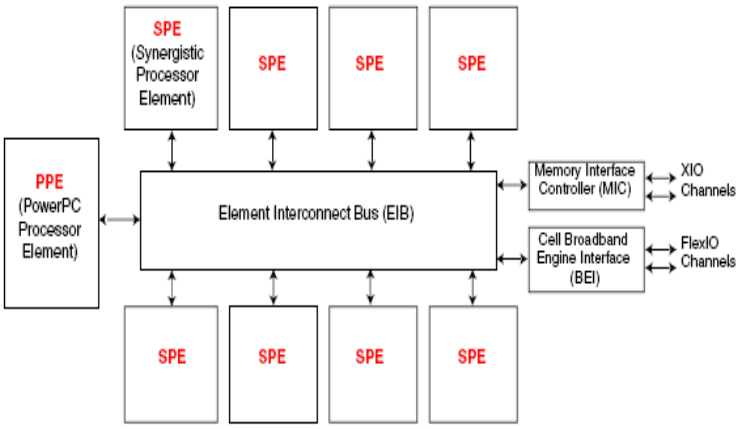

Figure 1: Cell Broadband Engine Processor

H.264 codec as a main benchmark while implementing optimizations, although other general benchmarks are also tested. We evaluated the performance of our compiler using those benchmarks. Results indicate that significant speedup can be achieved. In H.264 codec, we achieved about 21 % improvement in performance, and up to 59 % improvement in other cases. We also compared this result with the result of XLC for SPE, which showed that our compiler outperforms in most cases.

This paper is organized as follows. Next section discusses on the background. In section 3, we describe an in depth description of the SPE specific optimizations. Experimental results are discussed in section 4, followed by related work in section 5. Finally section 6 concludes the paper.

2. Background

The CBE is a relatively new architecture, which uses multilevel heterogeneous parallelism in order to achieve maximum performance of applications. It is designed with such applications in mind as games and media streaming applications which have highly parallel code such as game physics and image processing, providing both flexibility and performance to them. The CBE is composed of 1 PowerPC Processor Element (PPE) and 8 Synergistic Processor Elements (SPEs). The PPE have two levels of globally-coherent cache and is designed to run operating system and to control task distribution between 8 SPE elements.

On the other hand, the SPE is designed to run computation intensive, small, and preferably control reduced tasks. An SPE consists of a SIMD processor having 128 128-bits register file, local non-coherent memory, and globally-coherent DMA engine. Nearly all instructions provided by the SPE operate in a SIMD fashion on 128 bits of data representing either 2 64-bit double float or long integers, 4 32-bit single float or integers, 8 16-bit shorts, or 16 8-bit bytes. An SPE have two pipelines - even pipeline for arithmetic and logic computation, and odd pipeline for load/store and branch instruction. Two pipelines can issue instructions at the same time if the instructions are independent. Because the SPE do not have branch prediction hardware (instead provides branch hint instruction) and pipeline is very deep, care must be taken not to cause severe branch penalty (18 cycles).

The SPE's 256K bytes local memory (LS) supports 16-byte accesses for memory instructions and 128-byte accesses for instruction fetch and DMA transfers. Because the LS is single ported, instruction fetch, memory instructions, and DMA compete for the same port. Instruction fetches occur during idle memory cycles in 64 byte address alignment. A DMA engine is used to transfer data at high speed between an external memory and the SPE local memories, which should be explicitly initiated by programmers.

Our work is an attempt to address difficulties in developing efficient software on the CBE. The performance and flexibility supported by the CBE has very important meaning to devices like HDTV/IPTV, since it enables software implementations used in place where built-in hardware implementations have been used. For example, fast software codec can be plugged into the device for new media type, which means low cost and easy of maintenance. For this to happen, however, we need to fully utilize the performance capability of the CBE, especially the SPE.

Thread parallelization among the PPE and SPEs, and data/instruction parallelization in a SPE are our main concerns for performance, which we are trying to address by compiler technology. For thread parallelization, we've been developing OpenMP support for the CBE. Because most of performance of the CBE comes from SPEs, we designed so that most computations performed at SPE side. For efficient data transfer between SPEs and the PPE, we've been implementing software cache with memory optimization techniques like data prefetching. For small size of LS, we used overlay technology with a research on optimal partitioning of overlays going on. Although this project is based on GCC and implementation of its OpenMP support, we need to modify much because of some challenging features of the CBE such as heterogeneous processor type and explicit memory transfer.

3. Optimization for the SPE

There are two major challenges that must be overcome to exploit the performance provided by the CBE: parallelization and memory transfer. For SPE, we mainly focused on ILP, such as loop optimization and instruction scheduling. Data parallelization via auto-vectorization and thread level parallelism and memory transfer is started but not yet completed.

3.1. Swing Module Scheduling

Swing modulo scheduling (SMS), which is implemented in GCC, is a software pipelining approach which exploits ILP out of loops by overlapping successive iterations of the loops and executing them in parallel. The key idea software pipelining is to find a pattern of operations (kernel code) so that when repeatedly iterating over this pattern, it produces the effect of initiating an iteration before the previous ones have completed [15]. The number of cycles between initiations of successive iterations (i.e., the number of cycles per stage) in software pipelined loop is termed as Initiation Interval (II). The goals of SMS are to find near minimal II, register pressure, and stage count.

Swing Modulo Scheduling takes dependency graph as its input, and after performs ordering of nodes (instructions) in the graph by given priority, and then it schedules all the nodes in the graph using ordering information computed. Scheduling usually is successful if II is big enough, but II should be minimized for better performance. However, when it schedule a node which has its predecessors and successors are already scheduled, since it does not perform backtracking, blocking situations often take places which makes scheduling fails no matter how big is II.

In Figure 3, scheduling phase reads dependency graph as shown in the left and then schedules from V1 to V7 as shown in the right The schedule for each node is computed in such a way that a predecessor should be scheduled before its successor, but a node pointed to by a back-edge node (created by cross-iteration dependency) should be scheduled before that node by the amount of II. As shown in the right of Figure 3, however, this scheduling algorithm fails when it try to schedule V7 because of conflicting conditions. Such dependency graphs and blocking patterns are found very often in SPU programs due to the 16 bytes memory access size in SPU – `load/store` of two different memory addresses (i.e., adjacent elements of an array) with the same 16-byte aligned address can be dependent on each other. These patterns can also be caused by cross-iteration loop dependencies of common type, for example, in loop nests. We modify described scheduling rule as follows: if both predecessors and successors of node v are scheduled, then search direction should be based on successors or predecessors depending on whether only forward successors or forward predecessors were scheduled. This allows preventing blocking situations as described above. After described modification, SMS gained 25% improvements in performance when it is applied to the example code in Figure 2.

```
float a[100],b[100],c[100],d[100];

void foo (int n)
{
 int i,m;
 float x,y,z,t;
 for( i = 0; i<n; i++){
   x = a[i+l]; y = b[2*i + m];
   z = c[3*i + m];
   t = (x + y)/( x - y)*(x* y + z);
   d[i] = c[i+1] * t;
   d[i+1]= d[i]*( c[i+2]+d[i-1]);
  }
 }
```

Figure 2: Sample program code for SMS.

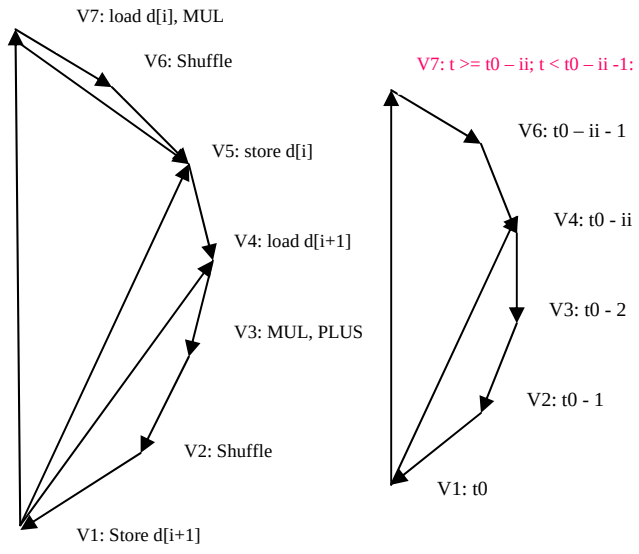

Figure 3: In the left is a schematic representation of dependency graph corresponding last 2 lines of the loop of the sample code in Fig. 2, and in the right is a simplified dependency graph with schedule information. Load is considered for the left-hand side of an assignment.

To get more performance, we integrated SMS with if-conversion so that conditional branches inside a loop removed and just one basic block (BB) remained in the loop. For this to happen, we've merged BBs in an innermost loop into a single BB without considering actual scheduling. Although we do not consider actual scheduling while merging, it generally shows good results since avoiding branch cost is more important in SPE and accurate instruction scheduling heuristics are still very local – does not consider cross-iteration ILP. After merging, CSE pass is run to remove redundancies comes from merging BBs, followed by SMS. Finally, we compared the schedule of instructions before and after this integrated SMS approach, and revert to the original loop body in case that is better. Although there is already a pass performing if-conversion before SMS in GCC, if-conversion itself cannot convert the body of a loop with internal control into a single BB, which makes SMS impossible.

Even though SPE does not support predication except for `selb` instructions, this integration has shown good results. Since SPE does not have memory access exceptions, it is possible to speculate `load` and `store` instructions while if-conversion. Store speculation is possible as follows:

```
if(cond) *p = x ⇒
                  val = *p;
                  y = selb(x,val,cond);
                  *p = y
```

3.2. Dependency analysis

SPE's memory instructions support only 16-byte aligned and 16-byte wide accesses. In case of accessing scalar variable, compilers must split a scalar load operation to a load-rotate instruction chain, and a scalar store to a load-shuffle-store chain. Because of these splits, two different memory addresses with the same 16-byte aligned address can be dependent on each other. The serial execution of the split instruction chain causes many stalls in SPE pipelines due to dependencies coming from the same operands of the instructions. Thus, we need precise dependency analysis confirming the independency between actually independent operands of different chains. In order to precisely break as many fake dependencies as possible for the SPE, we improved routines for memory dependency check.

First of all, we checked whether two memory references overlap by analyzing the dependency considering SPE's 16-byte access. Specially, in case of elements compactly laid in an array or a struct (or union), we exactly extracted the aligned base addresses, offsets from base, and access sizes. Then we checked whether two memory references can access same 16-byte aligned memory range. Additionally, in case that we can not find the aligned bases due to elements accessed via pointers, we compared the type of their contexts, and checked whether an element can be structurally isolated from the other by the amount of 16 bytes.

In case that we can find the distance between two memory references, basically we need to check whether the distance is greater than the access size of the preceding references, which means independency. Since the access size in SPE is 16 bytes, we modified the independency condition to check whether the distance is greater than 16 bytes.

The original GCC has used only the type-based alias analysis in order to break the dependencies between two memory references including pointers. For better analysis, we propagated the points-to sets including inter-procedural as well as intra-procedural analysis to RTL (low-level IR in GCC) optimization phases, and then checked whether two points-to sets intersect each other.

3.3. Other SPE specific optimizations

SPEs have deep pipelines, which make the penalty of incorrect branch prediction high, namely 18 cycles. Also, because branches are only detected late in the pipeline at a time where there are already multiple fall-thru instructions, SPE simply assumes all branches to be not-taken. Thus, we first tried to eliminate taken branches by if-conversion. As shown in the previous sub-section, we aggressively applied if-conversion even inside loop optimization.

In cases taken branches cannot practically be eliminated, such as function calls or loop closing branches, we used a branch hint instruction (refered to as `hbr`) which is provided by SPE. This instruction specifies the location of the branch and its likely target address. When a hint is correct and scheduled at least 11 cycles before its branch, the branch latency is actually 1 cycle. When a hint is scheduled at least 4 instruction pairs, there is some delay before the branch is performed but still no branch penalties apply. Other cases, normal branch penalties occur.

To improve performance by allowing more dual-issue and less instruction-fetch latency, we used bundler. We implemented bundler in such a simple way that if one instruction should start a new cycle or elements of two consecutive instructions are not on

their proper pipelines, `nop` instruction is added to the instruction sequence to make dual issue. Also, if the number of instructions between branch hint and its branch is smaller than that for avoiding branch penalty, other instructions or `nop`s are scheduled in between. To reduce number of instruction fetches, we aligned function body and innermost loop nest in 64 byte alignment which is unit of instruction fetch in SPE.

We basically reused what Sony has implemented in GCC4 for SPE – branch hint and bundling [13]. Then we improved branch hint to be added for every branch (for innermost loop, outer loops are sometimes ignored though) with no branch penalty, bundling to make more dual issue. If-conversion, branch hint, bundling and loop optimization is integrated for orchestration of each optimizations, and innermost loop body and function body is aligned for better instruction fetch.

3.4. Auto-vectorization

Since the SPE is a SIMD unit, converting normal statements in a vector form is essential for improving performance. For automatically generating SIMD code, we used auto-vectorization support from GCC [10, 12]. It supports vectorization of such important statement as reduction, unaligned-access of arrays, and stridden-access of arrays, so what we need to do is to define some target instructions and hooks for the SPE. Because GCC's auto-vectorization is based on loop, however, large basic-blocks with no loop which are usual in codec cannot gain much performance from it. To address this problem, we are going to implement superword-level parallelism (SLP) which can vectorize consecutive instructions in a large BB [11].

3.5. Profiling support

GCC supports profile-directed optimization by automatically add instrumentation code in executable files. GCC adds instrumentation code to count edge probability and stored it in a counter table for a source file. The size of LS in SPE is only 256KB, however, so instrumented programs for profiling often does not fit into LS size. To address this problem, we tried to minimize the size of instrumentation code and counter table to be fit in LS. In our approach, only one source file is compiled with profile-generate option. If there are N source files, minimum N executables are generated. To minimize instrumentation code size, we limit the number of edges to be instrumented. If number of edges (`E`) in a source file is greater than the limit (`MAX_INSTR_EDGES`), `M = ceiling(E / MAX_INSTR_EDGES)` executables is generated for the source file. After running `M` executables, complete profile data for the source file is made. The profile data files can be used when compile the source files again.

When the instrumented functions are invoked in a SPE, it sends signal to the PPE with command. Then the PPE gets parameters via DMA, and then save the result of profile information.

4. Experimental Results

We evaluated the performance of our compiler using well-known benchmarks and a H.264 decoder running on a CBE processor (BladeCenter QS20 from IBM).The optimizations described in the preceding sections were all applied. Because all benchmarks are already (manually) simdized enough, we gained only small improvements from auto-vectorization. Instead, much of our improvements came from SPE specific optimizations (branch hint and bundling), swing modulo scheduling, if-conversion, and precise dependency analysis.

Table 1: Speedups of our compiler over H.264 codec compared with original GCC integrated with Sony's optimization. Both are compiled with maximum optimization level.

Decoder Module	Optimized Cycle	Original cycle	Speedup
CAVLD	11,516	14,726	1.28
Deblock	7,766	8,837	1.14
MC	5,015	5,870	1.17
IntraP	298	352	1.18
IQIT	1,010	1,258	1.25
Total	**25,606**	**31,044**	**1.21**

Table 2: Speedups of our compiler over general benchmarks compared with original GCC integrated with Sony's optimization. Both are compiled with maximum optimization level.

Name	Optimized Cycle	Original cycle	Speedup
matrix_mul	5,651,771	5,550,991	0.98
oscillator	2,910,565	3,152,589	1.08
vse_subdiv	5,060,659	5,113,267	1.01
Raytrace	29,439,515	28,512,762	0.97
fft_2d	2,547,289	2,623,213	1.03
LU	3,526,192	3,536,415	1.00
convolution	27,800,091	26,480,317	0.95
ip checksum	1,448,327	2,308,622	1.59
linpack single	2,266,411	2,438,692	1.08
linpack double	2,532,974	2,737,000	1.08
whetstone	962,588	1,040,624	1.08
dhrystone	76,551,766	81,701,761	1.07
IDEA	1,142,819	1,047,790	0.92
huff_enc	1,660,913	1,922,909	1.16
gzip	839,807	953,740	1.14

Overall, we achieved 21% increases in performance over H.264 codec, up to 59% over other general benchmarks. The main reason our optimization achieved better performance in codec than other benchmarks is that basically we focused our optimization for codec, thus used codec as our main benchmarks for optimizations. Generally we achieved better performance than XLC in most cases, as shown in table 3.

Table 3: Comparison of results of our compiler with that of XLC. Both cases are optimized with maximum optimization level.

Decoder Module	Optimized Cycle	XLC cycle	Speedup
CAVLD	11,516	15,947	1.38
Deblock	7,766	9,176	1.18
MC	5,015	6,062	1.21
IntraP	298	464	1.56
IQIT	1,010	1,268	1.26
Total	**25,606**	**32,917**	**1.29**

One of our major concerns when applying optimizations such as loop optimization is the increase in code size. In case we added branch hints for every branch with no branch penalty (by adding 4 instruction pairs including `nop`) and performed aggressive bundling, we've got 10% increases in code size, which leads us to use this when performance is really important than code size.

5. Related works

IBM and Sony, which had participated in development of the CBE, introduced and have implemented compilers for it. IBM provides XL C/C++ Alpha Edition Compiler [5, 9] tuned for CBE. XL C/C++ implemented SPE specific optimizations such as bundling for dual issuing, compiler assisted branch prediction, and instruction fetch. It addressed data parallelization via automatic generation of SIMD codes. It also supports task-level parallelization via code and data partitioning across SPEs, though it is not available at the time of writing. Our project is similar to this, but our compiler is not general but specific for some target application – media processing workloads, especially codec.

Sony provides tool chains including bin utilities, new libraries, and compilers for the CBE. Sony's compiler is based on GCC open-source compiler. Sony ported it to the SPE and added SPE specific optimizations. As its compiler adopted the versioned-up GCC, the ported compiler could naturally inherit the new compiler optimization techniques, then present good performance in general benchmarks. However, it does not support OpenMP yet, though they have plan [13]. We used some instruction scheduling optimization from Sony, but improved more as shown in section 3.3.

RapidMind provides the software development platform [7] for data parallel programming on the CBE, the GPUs, and multi-core CPUs. The RapidMind platform includes a dynamic compiler and runtime management system for parallel processing. In case of the CBE, the RapidMind platform allows C++ familiar programmers to easily make parallel programs distributing the computations across SPEs without any need to manipulate the low level details.

6. Conclusions and Future Works

In this paper, we have presented our compiler optimizations and approaches to exploit heterogeneous parallelism found in the CBE architecture. Our compiler implements various optimizations. For ILP,

we implemented SPE specific optimization, loop optimizations, and instruction scheduling with precise dependency analysis for the SPE. For data parallelization, we ported GCC autovect branch for the SPE and improved a bit. As a result, we achieved 21% improvements in H.264 codec, and up to 59% in other benchmarks.

Though this project is currently functional, still it is far from its completion. For the SPE, we are going to implement more optimizations for better performance in multimedia and graphics workload. We found many large BBs in codec, and if-conversion also makes large BBs. Thus, we are going to implement SLP to address large BBs by data parallelization. Furthermore, we are going to implement global scheduling support for better utilize ILP. For better utilization of the CBE, we are working on thread parallelization including partition, data prefetch, and DMA scheduling for our OpenMP implementation.

Our main target for optimization at the time of writing is codec (i.e., H.264 and Mpeg2), thus we are going to use various codecs for our benchmarks. Finally, we are going to use our compiler for HDTV/IPTV and portable media devices, especially for downloadable codec.

References

[1] Donald Tanguay, et al., "Nizza: A framework for developing real-time streaming multimedia application", *http://www.hpl.hp.com/techreports/2004/*

[2] Karl Czajkowski, et al., "A resource management architecture for meta-computing systems", *Lecture Notes In Computer Science*, *Vol. 1459*, 1997

[3] Seiji Maeda, et al., "A Cell software platform for digital media application", *COOL ChipsVIII*, 2005

[4] *http://msdn.microsoft.com/*

[5] *http://www-128.ibm.com/developerworks/power/cell/*

[6] *http://www.cradle.com/*

[7] http://www.rapidmind.net/

[8] Steven S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufman Publishers, 1997.

[9] Alexandre E. Eichenberger, et. al, "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture", IBM Systems Journal, VOL 45, NO 1, 2006.

[10] Dorits Nuzman, et. al., "Auto-vectorization of Interleaved Data for SIMD", In Proceedings of the Programming Language Design and Implementation (PLDI '06), Jun 2006.

[11] Jaewook Shin, et. al., "Exploting Superword-Level locality in Multimedia Extension Architecture", Journal of Instruction-Level Parallelism 5 (2003) 1-28

[12] Dorit Nuzman, et. al., "Multi-platform Auto-vectorization", In Proceedings of the International Symposium on Code Generation and optimization (CGO '06), 2006,

[13] Ulrich Weigand, "Porting the GNU Tool Chain to the Cell Architecture", In Proceedings of the GCC Summit 2005.

[14] "Cell Broadband Engine Programming Handbook", http://www-128.ibm.com/developerworks/power/cell/docs_documentation.html

[15] Josep Liosa, et.al., "Lifetime-Sensitive Modulo Scheduling in a Production Environment", IEEE Transactions On Computer, VOL. 50, No. 3, March 2001.

[16] http://www.mc.com/cell/

Invited Talk:

GCC-based CLI Toolchain for Media Processing

Roberto Costa, roberto.costa@st.com
Andrea C. Ornstein, andrea.ornstein@st.com
Erven Rohou, erven.rohou@st.com
Andrea Bona

STMicroelectronics Manno lab

Abstract

A major cost in the development of software for multimedia processing is due to the high variability of target processors/accelerators and to the speed to which existing platforms evolve and new ones are introduced. Maintaining compilers and toolchains for all the products is a burden; in addition, software development must keep up with the fast pace of the market in order not to risk that, by the time software is ready, a product has become obsolete.

This presentation describes an unified development toolchain based on GCC using CLI binaries as the intermediate format and it shows the advantages of such a solution. More precisely, compilation is split into a platform-independent step that compiles source code into CLI-compliant binaries and a platform-specific step that natively compiles the deployed CLI executables just in time or ahead of time. The platform-independent compilation step is performed by GCC with a CLI back-end, conceived and developed on purpose by a small team in STMicroelectronics.

Empirical Auto-tuning Code Generator for FFT and Trigonometric Transforms

Ayaz Ali and Lennart Johnsson
Texas Learning and Computation Center
University of Houston, Texas
{ayaz,johnsson}@cs.uh.edu
Dragan Mirkovic
MD Anderson Cancer Center
The University of Texas, Houston
dmirkovi@mdanderson.org

***Abstract*—We present an automatic, empirically tuned code genenrator for Real/Complex FFT and Trigonometric Transforms. The code generator is part of an adaptive and portable FFT computation framework - UHFFT. Performance portability over varying architectures is achieved by generating highly optimized set of straight line C *codelets* (micro-kernel) that adapt to the microprocessor architecture. The tuning is performed by generating several variants of same size *codelet* with different combinations of optimization parameters. The variants are iteratively compiled and evaluated to find the best implementation on a given platform. Apart from minimizing the operation count, the code generator optimizes for access pattern, register blocking, instruction schedule and structure of arrays. We present details of the optimizations conducted at several stages and the performance gain at each of those levels. We conclude the paper with discussion of the overall performance improvements due to this aggressive approach to generating optimized FFT kernels.**

I. Introduction

The large gap in the speed of processors and main memory that has developed over the last decade and the resulting increased complexity of memory systems introduced to ameliorate this gap has made it increasingly harder for compilers to optimize an arbitrary code within palatable amount of time. The challenge to achieve high efficiency is now becoming more complex through the emergence of multi-core architectures and architectures using "accelerators" or special purpose processors for certain tasks, such as FPGAs, GPUs or the Cell Broadband Engine. To address the challenge to achieve high efficiency in performance critical functions, domain specific tools and compilers have been developed. There is a need for the development of high performance frameworks that aid the compilers by generating architecture conscious code for performance critical applications.

The Fast Fourier Transform (FFT) is one of the most widely used algorithms for scientific and engineering computation especially in the field of signal processing. Since 1965, various algorithms have been proposed for solving FFTs efficiently. However, the FFT is only a good starting point if an efficient implementation exists for the architecture at hand. Scheduling operations and memory accesses for the FFT for modern platforms, given their complex architectures, is a serious challenge compared to BLAS-like functions. It continues to present serious challenges to compiler writers and high performance library developers for every new generation of computer architectures due to its relatively low ratio of computation per memory access and non-sequential memory access pattern. FFTW[7], [6], SPIRAL[13] and UHFFT[12], [11] are three current efforts addressing machine optimization of algorithm selection and code optimization for FFTs.

Current state-of-the art scientific codes use re-usable components and a layered scheme to adapt to the computer architecture by using run-time performance analysis and feedback[14], [15]. At the lower level, the components may use highly optimized sets of straight line parametrized codes (micro-kernels) that are generated to adapt to microprocessor architecture. At the higher level, the parametrization allows for optimal data access patterns to be searched to enhance effective memory bandwidth and lower latency without extensive code optimization. In UHFFT[12], [11], the adaptability is accomplished by using a library of composable blocks of code, each computing a part of the transform. The blocks of code, called *codelets*, are highly optimized and generated by a code generator system called FFTGEN. We use an automatic code generation approach because hand coding and tuning the DFT is a very tedious process for transforms larger than size five. Moreover, by implementing a number of optimizations, we were able to achieve operation counts that were smaller than traditionally assumed for the transform for many sizes.

As shown in Figure 1, eleven different formulas for FFT size 12 result in three different floating point operation counts. In the given graph and the results to follow, we use (Million Floating Point Operations per Second) "MFLOPS" metric to evaluate the performance. We use standard *radix-2* FFT algorithm complexity to estimate the number of floating point operations and then divide that by the running time in micro seconds.

Related Work

Using simple heuristics that minimize the number of operations is not sufficient to generate the best performance FFT kernels; especially on modern, complex architectures.

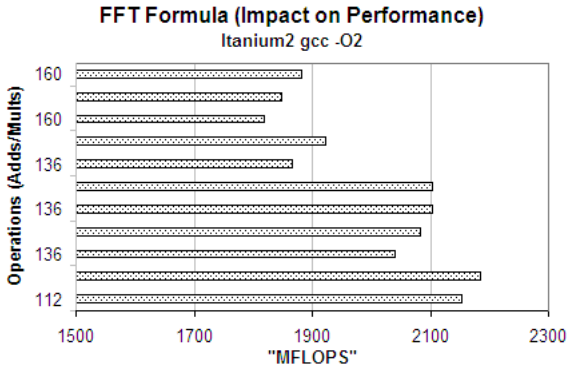

Figure 1. Performance vs Operation Count for different formulas of size 12 Complex FFT.

Among other factors, instruction schedule and pipelining play an important role in the overall performance. For embedded systems with limited number of registers and instruction cache, it is even more important to employ aggressive optimization techniques to make best use of available resources.

In [9], authors describe an iterative compilation approach to exploring the best combination of tuning parameters in an existing compiler. The empirical technique is shown to yield significant performance improvement in linear algebra kernels. However, the optimizations are mainly focused on loops, which reduces the prospects of major performance gain in small FFT code blocks. More recently, the iterative empirical compilation techniques have been studied [5], [8], [4] on whole applications instead of compute intensive kernels.

An excellent example of automatic generation of tuned linear algebra micro-kernels is given in [14], [15]. The methodology, called Automatic Empirical Optimization of Software (AEOS), employs iterative empirical evaluation of many variants to choose the best performing BLAS routines.

Because of the unique structure of FFT algorithms with output dependence between successive ranks (columns), loop level optimizations are not as effective as in the case of linear algebra codes. The most important tuning parameter for FFT turns out to be memory (register and cache) blocking and the factorization (formula). SPIRAL[13] searches for the best FFT formula for a given problem size using empirical evaluation and feedback. FFTW[7], [6] employs simple heuristics, similar to the one given in section 3, to determine the best formula. Unlike SPIRAL, both FFTW and UHFFT[12], [11] generate micro-kernel of straight line FFT code blocks (*codelets*) and defer the final optimization of a given problem till the run-time. The set of *codelets* (micro kernel), generated at installation time is usually restricted to sizes depending on the size of instruction cache and the number of floating point registers. For some architectures, FFTW also generates ISA specific *codelets* to boost the performance. We believe that such translation should be performed by compiler and the code generator should aid the compiler to achieving that goal. This paper presents an aggressive approach to generating adaptive and portable code for FFT and Trigonometric Transforms by iteratively compiling and evaluating different variants. Apart from exploring the best FFT formula and register blocking, we try limited number of instruction schedules, translation schemes to probe and adapt to both architecture and compiler.

This paper is organized as follows. Section 2 gives the design details and functionality of the code generator. Section 3 describes the automatic optimization and tuning methodology using compiler feedback loop in the UHFFT. Finally, in section 4, we report performance gain due to our new approach and develop models to understand the cache performance of *codelets* for different strides.

II. Design Details

The UHFFT system comprises of two layers i.e., the code generator (FFTGEN) and the run-time framework. The code generator generates highly optimized straight line C code blocks called *codelets* at installation time. These *codelets* are combined together by the run-time framework to solve large FFT problems on Real and Complex data. The type of code to generate is specified by the type and size of the problem. The code generator adapts to the target platform i.e., compiler and hardware architecture by empirically tuning the *codelets* using iterative compilation and feedback. In order to limit the search space of various parameters, tuning is conducted in three phases (stages) and at the end of each phase, values of the parameters are selected. The design overview of FFTGEN is given in Figure 2. FFTGEN2, which implements the new empirical tuning strategy will be integrated in UHFFT version 2.0.1; the beta version is available online and can be downloaded at [1].

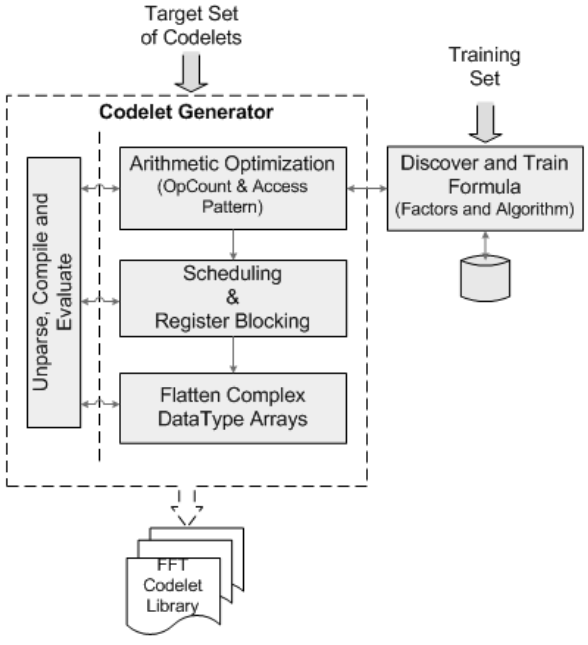

Figure 2. Fftgen2 Block Diagram

A. *Specifying the Set of* Codelets

The set of desired *codelets* can be defined in a script file, which internally uses FFTGEN2 to generate the *codelets*.

The *codelet* sizes should typically be limited by the size of instruction cache and the number of registers on target architecture. After the set of desired *codelet* sizes is specified in the script file, code generator does not require any further user intervention in order to produce highly optimized micro-kernel of *codelets* for the platform. Nevertheless, an expert user can suggest alternative FFT formulas (factorizations) that FFTGEN2 should try as variants of desired *codelet*. We have implemented a concise language called *FFT Formula Description Language* (FFDL), which is used to describe the FFT factorizations. The code generator supports a subset of FFDL rules as given in Figure 3. Full FFDL specification is part of the UHFFT 2.0.1 run-time framework, which includes multiprocessor and multi-dimensional FFT expressions [3].

#	Productions
1-4	FFT → Module | FFTMR | FFTSR | FFTPF
5-6	FFTPF → FFTPF ***pfa*** Module | Module
7-9	Module → **[*rader*, FFT] *z*** | **[FFT] *z*** | Codelet
10	FFTMR → FFT ***mr*** Module
11	FFTSR → **[*sr* Module, Module] *z***
12	Codelet → **{2,3,4,…,16,32,64 …}**
13	z → $\mathbb{Z}$

Figure 3. FFT Formula Description Language (FFDL) Rules.

B. Types

The code generator (FFTGEN2) is capable of generating various types of tuned code blocks. As shown in Figure 4, each *codelet* is identified by a string of characters and size. For a given size and rotation, many different types of *codelets* can be generated depending on the following parameters:

Precision: *Codelets* with either double or single precision can be generated depending on this flag.

Direction: *Codelets* with both forward and inverse direction for FFT (complex/real) and DCT/DST Type-I can be generated.

Twiddle: Cooley Tukey FFT algorithm involves multiplication with diagonal twiddle matrix between the two FFT factors. Similar to the approach in FFTW[7], we generate special *twiddle codelets*, which have this multiplication fused inside to avoid extra loads.

I/O Stride: Every *codelet* call has at least two parameters, i.e. input and output vectors. The vectors can be accessed with different strides or same strides as in case of inplace transform. If the strides are same, excess index computation for one of the strides can be avoided by generating a *codelet* that takes only one stride parameter.

Vector Recurse: In most cases multiple vectors of same size need to be transformed. A *codelet* with the vector flag enabled lets user specify the strides (distance) between successive vectors.

Transform: Transform type is specified at the sixth position in the string. Apart from generating FFT and *rotated* (PFA) *codelets*, trigonometric transforms can also be generated by specifying flags '*c*' and '*s*' for DCT and DST transforms respectively.

Datatype: Both real and complex data *codelets* can be generated depending on the transform type.

Rotation: This parameter is only applicable to *rotated* (PFA) *codelets* that are used as part of the Prime Factor Algorithm. Note that a PFA *codelet* with rotation 1 is same as a NON-PFA *codelet*.

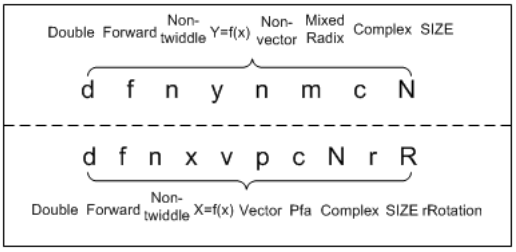

Figure 4. Two example strings for specifying the type of *Codelet* to be generated by FFTGEN2.

An illustration of two codelet types is given in Figure 4. *Rotated codelets* are useful as part of the PFA execution. Only a few options are applicable to PFA and DCT/DST; for instance, there is no need to generate separate inverse and forward direction *codelets* for PFA execution. Also the *twiddle* flag is ignored except when the *codelet* transform type is FFT (without rotation).

C. Fast Trigonometric Transforms

Trigonometric Transforms have a wide range of applicability, both in signal processing and in the numerical solution of partial differential equations. As given in [10], we have implemented FFT based algorithms for these transforms that require $2.5m \log m$ flops, assuming that m is a power of two. The basic idea is to expand the input vector x to $\tilde{x}$, as given in Table I, using certain symmetries, apply real FFT $F_{2m}\tilde{x}$ of size $2m$ and then exploit the symmetries to generate the *codelet* of complexity $\sim 2.5m \log m$. In our current version we have implemented only Type-I DCT and DST. Extending it to other types of transforms is straightforward for algorithms that are based on FFT.

III. Compiler Feedback Loop

Performance of the *codelets* depends on many parameters that can be tuned to the target platform. Instead of using global heuristics for all platforms, the best optimization parameters are discovered empirically by iteratively compiling and evaluating the generated variants. The optimization is performed in three stages (levels) independently; for example, all factorization policies are not tried for all block sizes. The evaluation module benchmarks the variants and returns an array of performance numbers along with the index of best

Table I
DCT/DST - INPUT VECTOR EXPANDED TO USE FFT ($m = 4$)

Transform	x	$\tilde{x}$
DST	$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}$	$\begin{bmatrix} 0 & x_1 & x_2 & x_3 & 0 & -x_3 & -x_2 & -x_1 \end{bmatrix}$
DCT	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 \end{bmatrix}$	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_3 & x_2 & x_1 \end{bmatrix}$
DST-II	$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}$	$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & -x_4 & -x_3 & -x_2 & -x_1 \end{bmatrix}$
DCT-II	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \end{bmatrix}$	$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & x_3 & x_2 & x_1 & x_0 \end{bmatrix}$

performing variant. In order to generate statistically stable measurements, each *codelet* is executed repeatedly. *Codelet* are normally called with non-unit strides in the context of a larger FFT problem. Therefore, it's performance should take into account the impact of strided data access. To achieve that goal, the benchmarking data is collected for strides 1 to 64K with increments of 2. We use average over all the samples to represent the quality of a variant. However, the selection policy can be easily extended to incorporate more complicated models.

Level 1: Arithmetic Optimizations

Different FFT formulas are evaluated in this phase to optimize for the floating-point operations and the access pattern. This phase generates the *butterfly computation*, which is abstracted internally as a list of expressions. Simple optimizations such as constant folding, strength reduction and arithmetic simplifications are applied on the list of expressions to minimize the number of operations.

Due to exponential space of possible formulas for a given size, built-in heuristics along with limited user supplied formulas (training set) are tried. Following is the algorithm that is used to generate different formulas that will eventually be evaluated to select the best.

Algorithm 1 FFT Factorization and Algorithm Selection

If $N = 2$ then $algo \leftarrow DFT$ and $r \leftarrow 2$
Else if $IsPrime(N)$ then $algo \leftarrow RADER$ and $r \leftarrow N$
Else
  $FindClosestBestSize\ n$ in Trained Set S
  If $n = N$ then then $algo \leftarrow S[n].algo$ and $r \leftarrow S[n].r$
  Else /* Use Heuristics */
    If n is a factor of N then $k \leftarrow n$
    Else $k \leftarrow GetMaxFactor(N)$
    If $gcd(k, \frac{N}{k})$ then $algo \leftarrow PFA$, $r \leftarrow k$
    Else if $N > 8 \,\&\, 4 \mid N$ then $algo \leftarrow SR$, $r \leftarrow 2$
    Else if $N > k^3$and $k^2 \mid N$ then $algo \leftarrow SR$, $r \leftarrow k$
    Else $algo \leftarrow MR$ and $r \leftarrow k$
  If $FactorizationPolicy \neq LeftRecursive$ then $r \leftarrow \frac{N}{r}$

The algorithm given above is called each time the factorization or algorithm selection is to be made. In the simplest case when N is equal to 2 or when N is a prime number, the algorithm returns the size N as a factor, selecting appropriate algorithm. If the size of *codelet* can be factorized then the training database is queried to find the best and closest size (n) to N such that either n is a factor of N or vice versa. If there is no such record found in the database, heuristics are used that minimize the operation count by preferring algorithms with lowest complexity. Note that when the training knowledge is utilized, the factors are selected in a *greedy* fashion, i.e. the factor with the highest performance is selected.

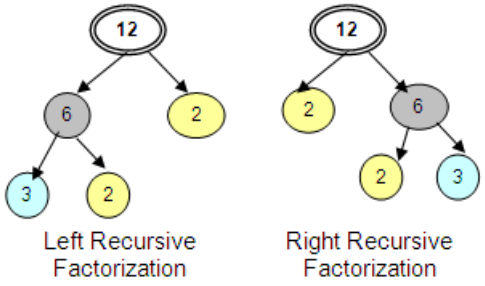

Figure 5. Examples of Left and Right Recursive Factorization Policies.

Table II
OPTIMIZATION LEVEL 1 TUNING PARAMETERS

Variants	Left Recursion	Rader Mode
1-3	0	CIRC,SKEW,SKEWP
4-6	1	CIRC,SKEW,SKEWP

In this phase, six variants are generated depending on the factorization policy, i.e. left recursive or right recursive factorization tree as illustrated in Figure 5. For each type of factorization policy, different rader algorithm options (depending on the convolution solver) are enabled as listed in Table II.

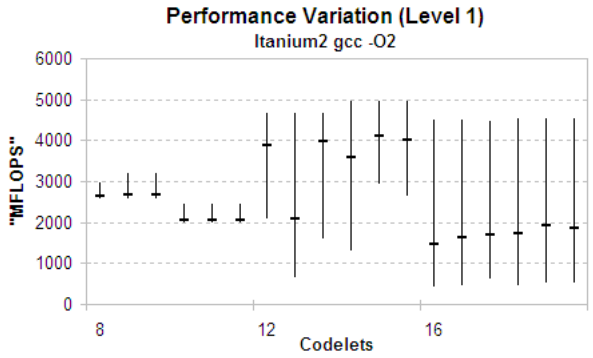

Figure 6. Level 1 Variants for Real FFT *Codelets* of size 8, 12 and 16. Six variants are evaluated for each codelet for varying strides given by vertical lines. The mean of performance for different strides is used to represent the performance of that variant.

Figure 6 shows performance variation for six variants of each of the three Real FFT *codelets* (8, 12 and 16). Each

vertical bar represents the minimum and maximum performance for that codelet depending on the stride. Notice that rader mode does not bring the performance variation to sizes that are powers of two. Hence only two variants need to be evaluated for powers of two FFTs, ignoring the rader mode parameter.

Level 2: Scheduling and Blocking

Second phase performs scheduling and blocking of expressions by generating a Directed Acyclic Graph (DAG). The main purpose of this scheduling is to minimize register spills by localizing the registers within block(s). A study [2] conducted by the authors of this paper revealed that even for very small straight line *codelets* the performance variation due to varying instruction schedules could be as large as 7%. Finding the best schedule and blocking for all factorizations is a hard problem, hence, only a few possible permutations of instructions and block sizes are evaluated.

Table III
OPTIMIZATION LEVEL 2 TUNING PARAMETERS

Variants	Reverse	Blocking
1-3	0	2,4,8
4-6	1	2,4,8

Three different block sizes, i.e. 2,4 and 8 are tried in combination with the option of reversing independent instructions within that block, as given by Table III. Each block is topologically sorted internally in the end. In total, six different variants of schedules and block sizes are generated and the best is selected after empirical evaluation.

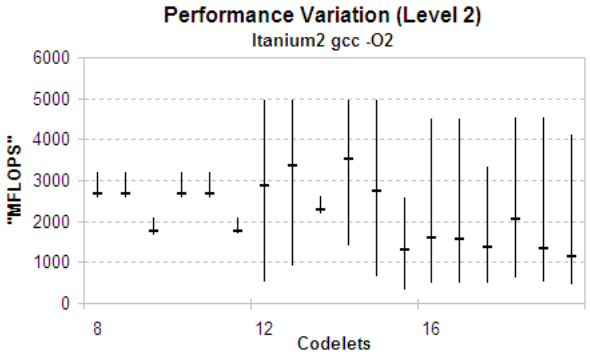

Figure 7. Level 2 Variants for Real FFT *Codelets* of size 8, 12 and 16. Six variants are evaluated for each codelet for varying strides given by vertical lines. The mean is used to represent the performance of a variant.

As shown in Figure 7, the performance variation at level 2 indicates that smaller block sizes perform better in most cases. There is no clear winner between the two sorting strategies of instructions. The reversing is intended to be a trial and error mechanism that generates some random permutations of instructions.

Level 3: Unparse scheme

All *codelets* take two essential parameters, i.e. input and output vector. For computation of FFT over complex data type, the vectors are represented by arrays of structures. Different compilers behave differently to the structure of input and output arrays. To generate a code that results in the best performance, we tried three different representations for input and output vectors of complex type as given in Figure 8. In the first representation, input and output vectors are accessed as arrays of structure (Complex). In the second scheme, each of the complex vectors is accessed as a single array of Real data type with the imaginary values at odd indices. In the third scheme, the complex vector is represented by two separate real and imaginary arrays of Real data type.

Figure 8. Input or Output Vector of complex data elements can alternatively be accessed as a single array of interleaved Real/Imaginary data or two arrays of Real and Imaginary data.

Table IV
OPTIMIZATION LEVEL 3 TUNING PARAMETERS

Variants	Scalar Rep.	I/O Vector Structure
First	0	Complex, 1 and 2 Real
Second	1	Complex, 1 and 2 Real

Apart from the three array translation schemes for Complex type *codelets*, two more variants are tried for all types of *codelets*, as given in Table IV. As shown in Figure 9, for small size codelets, we noticed that explicit step of replacing I/O vector elements in temporary scalar registers performed better in most cases. However it did increase the total size of code in terms of lines of C code.

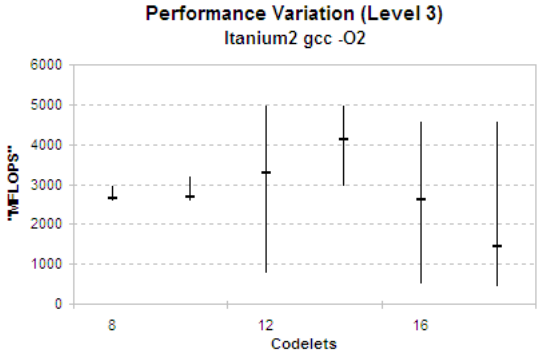

Figure 9. Level 3 Variants for Real FFT *Codelets* of size 8, 12 and 16. For Real Codelets, only two variants (based on the scalar replacement flag) are evaluated for varying strides given by vertical lines. The mean is used to represent the performance of a variant.

Table V shows the values of parameters selected after evaluating ten variants of Real FFT *Codelet* of size 16.

IV. RESULTS

We performed benchmarking of the complex type FFT *codelets* of sizes that were powers of 2 up to 128 for a range

Table V
SELECTED VARIANT FOR *Real* CODELET OF SIZE 16

Option	Selection
Left Recursion	On
Reverse Sort	On
Block Size	2
Scalar Replacement	Off
I/O Structure	One Real Vector

of input and output strides (also powers of 2). Since the sizes of cache lines, cache and memory are mostly equal to some power of 2, we expect to catch some of the worst performance behavior this way. Each reported data item is the average of multiple runs. This was done to ensure that errors due to the resolution of the clock are not introduced in the results presented. The benchmarking and evaluation was carried on two hardware architectures, Itanium2 and Opteron. A summary of the platform specifications is given in Table VI.

Table VI
ARCHITECTURE SPECIFICATIONS

	Itanium2	Opteron
Processor	1.5GHz	2.0GHz
Data Cache	16K/256K/6M	64K/1M
Registers	128 FP	88 FP
Inst. Cache	16K	64K
Associativity	4/8/12 way	2/16 way
Compilers	gcc3.4.6/icc9.1	gcc3.4.6/pathcc2.5

In the first set of benchmarks, the results were collected using two compilers for each of the two architectures. Note that same compiler and hardware architecture was used to generate the variants. We compared the performance of empirically tuned *codelets* with that of the *codelets* generated by previous version (1.6.2) of UHFFT to evaluate the efficacy of our new methodology. In the previous version of UHFFT, the *codelets* were generated using simple heuristics that reduced operation count without any blocking or scheduling.

As shown in Figure 10 and 11, there is significant performance improvement for large size Complex FFT *Codelets* for both compilers on Itanium 2. In most cases the performance improvement was seen for small as well as large strides as shown in the graph by vertical lines. Having said that, there is slight degradation of performance for size 16 *codelet*. We understand that the reason behind that could be our simple model for evaluating and selecting the best variant. As the stride is increased the performance of a *codelet* is dominated by the memory transactions which introduces noise in the evaluation of variants. As an alternative to choosing the variant with the best average performance over many strides, a *codelet* with best maximum and minimum performance could be chosen or the evaluation of variants could be limited to low strides so that cache misses are minimized.

In Figure 12 and 13, we performed the same exercise on Opteron using gcc and pathscale compilers. Even though we got some performance improvement for most sizes, the gain was not as much as on the Itanium 2. That may be due to the fact that Itanium 2 has 45% more FP registers than Opteron; thereby affecting the performance of large size *codelets.*

In the second experiment, we compared the performance

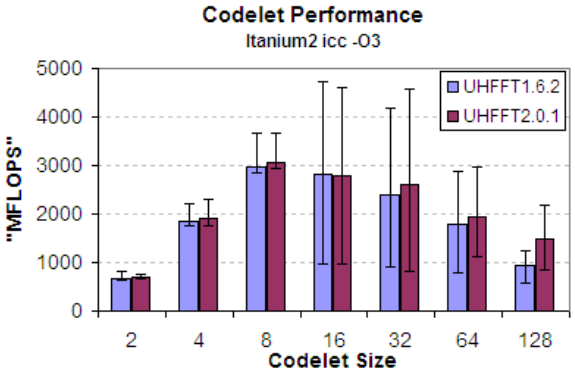

Figure 10. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Itanium 2 using icc. There is small performance improvement for most codelets. Each vertical error bar represents the variation in performance due to strides.

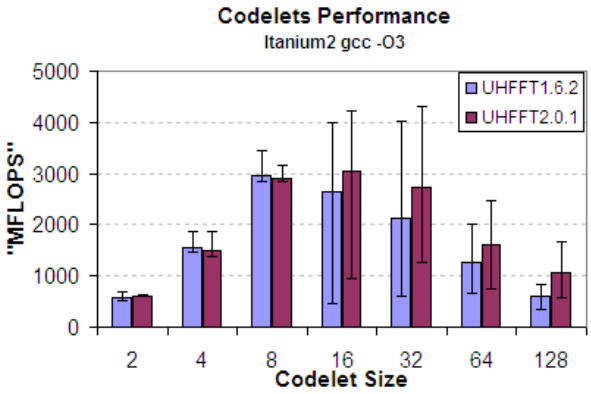

Figure 11. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Itanium 2 using gcc. There is significant performance improvement for larger size codelets. Each vertical error bar represents the variation in performance due to strides.

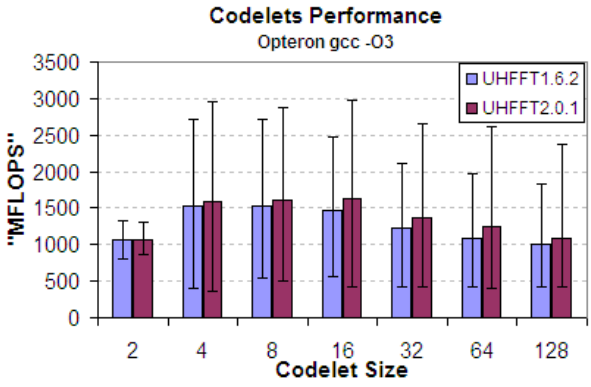

Figure 12. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Opteron using gcc.

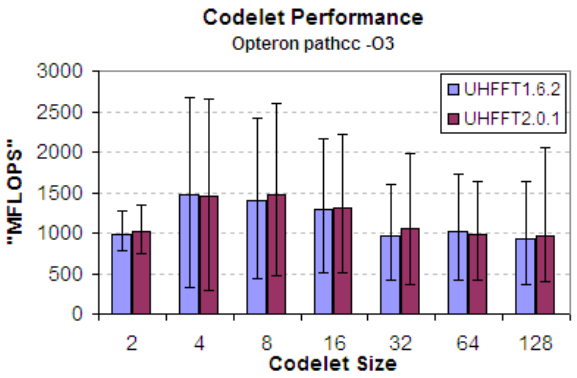

Figure 13. Performance Comparison of Complex FFT *Codelets* generated by UHFFT-1.6.2 and the new version 2.0.1 on Opteron using pathscale compiler.

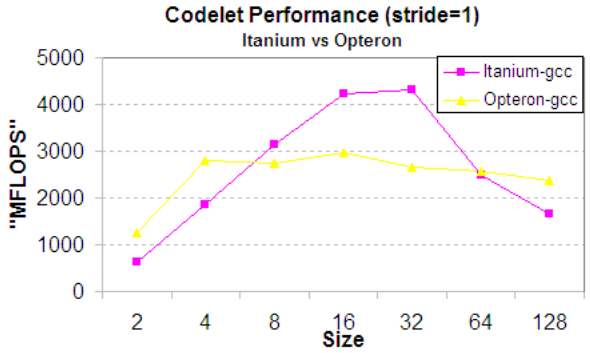

Figure 14. Impact of size of *codelet* on the performance. Larger codelets suffer from performance degradation due to register pressure and instruction cache misses.

of same *codelet* sizes for unit stride data on both Itanium 2 and Opteron. The performance of *codelets* increases with the size of transform and then starts deteriorating once the code size becomes too big to fit in the instruction cache and registers as shown in Figure 14. Interestingly, the performance decline on Opteron was not as sharp as found on Itanium 2. We believe, that is owing to the bigger instruction cache on Opteron compared to Itanium 2.

For all platforms considered, the performance decreases considerably for large data strides. If two or more data elements required by a particular *codelet* are mapped to the same physical block in cache, then loading one element results in the expulsion of the other from the cache. This phenomenon known as cache trashing occurs most frequently for strides of data that are powers of two because data that are at such strides apart are usually mapped to the same physical blocks in cache depending on the type of cache that is used by the architecture. On Itanium 2, the cache model shown in Figure 15 was harder to predict due to deeper hierarchy and more complex pipelining. However, as shown in Figure 16, for a more conventional architecture like Opteron, the sharp decrease in performance due to cache trashing occurs when:

$$size_{datapoint} \times stride \times 2 \times size_{codelet} \geq \frac{size_{cache}}{Associativity}$$

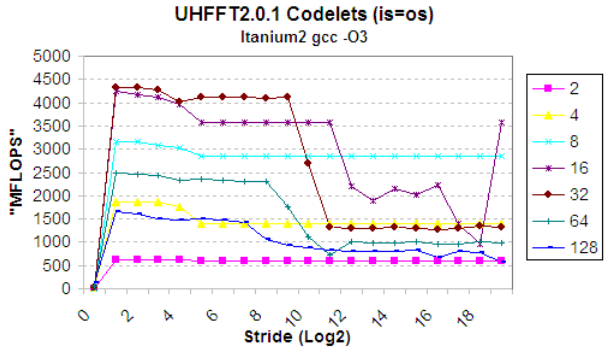

Figure 15. Cache Performance Model generated using Complex FFT *Codelets* with varying strides on Itanium 2.

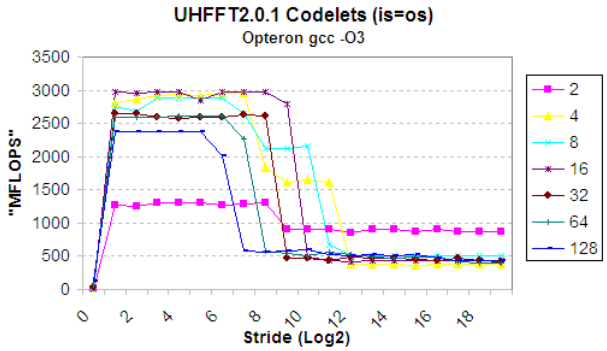

Figure 16. Cache Performance Model generated using Complex FFT *codelets* with varying strides on Opteron.

where datapoint size is the size of one data element (for complex data with 8 Byte real and imaginary data, each data point is of 16 Bytes), *codelet* size is the number of data elements being transformed by the *codelet*, cache size is the total size of the cache in Bytes, stride is the data access stride, and associativity is the type of cache being used by the architecture.

Conclusion

We have implemented and evaluated the empirical auto-tuning methodology in UHFFT library to generate highly optimized *codelets* for FFT(Real/Complex) and Trigonometric Transforms. The adaptive approach that we have chosen for the library is shown to be an elegant way of achieving both portability and good performance. Internally the code generator implements flexible mathematical rules that can be utilized to extend the library to generate other kinds of transforms and convolution codes, especially when the algorithms are based on FFT. The ease with which the whole UHFFT library tunes itself without user intervention allows us to generate any supported type of transform on any architecture.

References

[1] Uhfft-2.0.1 www.cs.uh.edu/ ayaz/uhfft. 2006.
[2] Ali, A. Impact of instruction scheduling and compiler flags on codelets' performance. Tech. rep., University of Houston, 2005.
[3] Ali, A. An adaptive framework for cache conscious scheduling of fft on cmp and smp systems. Dissertation Proposal, 2006.

[4] ALMAGOR, L., COOPER, K. D., GROSUL, A., HARVEY, T. J., REEVES, S. W., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2004), ACM Press, pp. 231–239.

[5] CHAME, J., CHEN, C., DINIZ, P., HALL, M., LEE, Y.-J., AND LUCAS, R. An overview of the eco project. In *Parallel and Distributed Processing Symposium* (2006), pp. 25–29.

[6] FRIGO, M. A fast fourier transform compiler. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (New York, NY, USA, 1999), ACM Press, pp. 169–180.

[7] FRIGO, M., AND JOHNSON, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

[8] FURSIN, G., O'BOYLE, M., AND KNIJNENBURG, P. Evaluating iterative compilation. In *LCPC '02: Proc. Languages and Compilers for Parallel Computers.* (College Park, MD, USA, 2002), pp. 305–315.

[9] KISUKI, T., KNIJNENBURG, P., O'BOYLE, M., AND WIJSHO, H. Iterative compilation in program optimization. In *Proc. CPC2000* (2000), pp. 35–44.

[10] LOAN, C. V. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[11] MIRKOVIC, D., AND JOHNSSON, S. L. Automatic performance tuning in the uhfft library. In *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I* (London, UK, 2001), Springer-Verlag, pp. 71–80.

[12] MIRKOVIC, D., MAHASOOM, R., AND JOHNSSON, S. L. An adaptive software library for fast fourier transforms. In *International Conference on Supercomputing* (2000), pp. 215–224.

[13] PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J., PADUA, D., VELOSO, M., SINGER, B. W., XIONG, J., FRANCHETTI, F., GAČIĆ, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation" 93*, 2 (2005), 232–275.

[14] WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2 (2001), 3–35.

[15] WHALEY, R. C., AND WHALEY, D. B. Tuning high performance kernels through empirical compilation. In *ICPP '05: In Proceedings of the 2005 International Conference on Parallel Processing* (Oslo, Norway, 2005), EEE Computer Society, pp. 89–98.

Enabling Word-Width Aware Energy and Performance Optimizations for Embedded Processors

Andy Lambrechts[1,2], Praveen Raghavan[1,2], David Novo[1,2], Estela Rey Ramos[3], Murali Jayapala[1], Francky Catthoor[1,2], Diederik Verkest[1,2,4].

[1]IMEC vzw, Belgium [2]Dept. of Electrical Engineering, KULeuven, Belgium
[3]University of Vigo, Spain [4]Dept. of Electrical Engineering, V.U.B., Belgium

arpa@imec.be

ABSTRACT

Modern mobile devices need to be extremely energy efficient. They need to be able to run demanding applications and support communication modes of increasing computational demand. To make this desired behavior possible within the battery-operated context, all parts of these embedded platforms need to be optimized and applications need to be mapped in the most efficient way. In this work we present an approach that enables designers to estimate the effect of exploiting word-width information of application data on the energy consumption and performance of the platform. A correct estimate can help to correctly evaluate optimizations that exploit word-width information. In order to estimate the effect of these word-width aware transformations, energy models have to be made sensitive to word-width information and simulation has to be refined from a component activation based approach to a more detailed dynamic range sensitive approach. Transformations based on word-width info can change the number of computational operations, the number of accesses to data and instruction memory hierarchy and hence affects the number of cycles needed to execute the application. The word-width aware energy modeling presented in this paper shows a difference in energy estimations accuracy of about 15 % for a representative example, when compared to a less accurate non-word-width aware estimation. We show how word-width information can be used during mapping, how the new energy models enable a correct evaluation of the gains and how a significant energy and performance improvement can be achieved using *Software SIMD* as an example optimization. The demonstrated gains are about 80 % compared to a mapping without SIMD and over 30 % compared to traditional *hardware SIMD*, both for energy and performance.

Keywords

Energy-Aware mapping, Low Power, Word-Width Aware, Compiler Optimizations, SIMD

1. INTRODUCTION

Modern consumers demand portable devices that provide functionality comparable to that of their non-mobile counterparts, but still having a long battery life. In order to achieve these ambitious goals, designers of embedded platforms have to optimize all parts of the system. At the same time an efficient mapping of the application code onto the platform is key to achieve the best possible energy efficiency for a given platform. A mapping can be considered to be efficient if it is using the available hardware to its full potential.

Exploiting application knowledge during mapping can lead to big gains, from the algorithmic level down to the implementation. In this paper, we present an approach that enables the usage of word-width information (application knowledge) to evaluate different mapping options in terms of energy consumption and performance. This application data word-width information can be obtained in three different ways, namely being the result of an analytical refinement of the algorithm (e.g. value propagation techniques), using profiling (simulation based approach) or using a hybrid. Automated tools have been developed to obtain this word-width information (e.g. [13]). During the fixed point refinement application knowledge of the designer and end requirements of the platform can be exploited to obtain a range of word-widths, that can even depend on specific use-cases (e.g. quality of wireless connection, or best possible audio quality, depending on current state of the battery of a wireless device) .

To enable designers to see the potential gains of using this extra info during mapping and enable them to achieve better energy efficiency and higher performance, simulation and estimation techniques should give more accurate and more detailed energy estimates. To make this possible, first of all word-width aware energy models are needed. Current energy models used in ISS energy estimation assume that hardware components (e.g. adders, multipliers) are always operating on data that fill the complete width of these components. When the data used in a certain algorithm are less wide, these components internally toggle less, which leads to a smaller energy consumption. Current processor and platform simulators can easily be extended to make use of this these more detailed energy models, once they are available. In this paper we give examples of such word-width aware models and how they can be generated. Using an example, we show how the improved accuracy of the energy estimation can influence a designer's decision or prevent wrong conclusions.

Word-width knowledge can be used in different ways within the scope of mapping, e.g. during scheduling, during assignment or to enable or guide optimization (see motivating ex-

ample in Section 2). In this paper we will show the potential gains this extra information can bring, by exploring different mapping alternatives and by using a technique called Software SIMD . Other optimizations based on word-width info are mentioned in Section 8.

This paper is organized as follows: Section 2 provides the motivation for this work. Section 3 presents the related work in the areas simulation, SIMD and Software SIMD. In Section4 the flow that is used to generate word-width aware models is explained. Section 5 shows the models that have been generated for the example platform that is presented in this paper and the improved accuracy when estimating the energy consumption. Section 6 shows, using an illustrative example, how this enabling work can improve the energy efficiency and performance during mapping, using a technique called Software SIMD. Section 7 summarizes the results and Section 8 mentions some ongoing research and future directions.

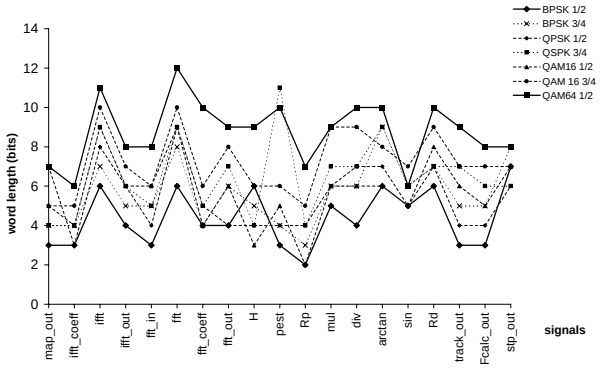

Figure 1: Minimum word-lengths for different baseband configurations (modulation scheme and coding rate) of an IEEE 802.11a/g WLAN transceiver

2. MOTIVATION

In this section we will motivate that varying word-widths are to be found in embedded applications, that these novel word-width aware energy models can improve the energy estimation accuracy when making use of this information and that this word-width information can effectively be used during mapping to improve energy efficiency and performance.

2.1 Word-Width Information

The enabling work that is presented here, allowing designers or compilers to exploit word-width information during application mapping, will be useful for applications that operate on data of different widths and in a context that requires extreme energy efficiency and a high performance. DSP designers often try to squeeze every last cycle out of a processor and every Joule has to be optimally put to use. In this context the usage of word-width information should be part of their optimization options. As exploiting word-width information during mapping can affect the number of accesses to memories, the impact on the overall system can be significant (as is shown in Section 5).

Figure 1 shows the result of a fixed point refinement performed for an IEEE 802.11a/g WLAN transceiver for different use-cases, depending on the modulation scheme (e.g. BPSK, Binary Phase-shift Keying etc.) and coding rate that are used. It can clearly be seen that a wide range of widths is available, within and across different configurations. Similar data type refinement steps can be found based on other design decisions (e.g. trade-off quality of the result against energy spent) and when other applications are mapped onto the same platform (e.g. audio, video and 3D applications), which leads to a larger diversity in word-widths that can be exploited. Because of this large diversity traditional hardware SIMD support is not cost-efficient (not all widths can be supported), which leaves room for other techniques to exploit this further.

2.2 Word-Width Aware Energy Models

The energy models that are currently used to get fast energy estimates using an ISS (Instruction Set Simulator) or simulators for embedded platforms provide average energy per activation numbers at a component level (Adder, ALU, Multiplier, Register File ,memories, etc.) and are *independent* of the actual data that are processed. Because of this abstraction a fast simulation is possible, but results do not allow any differentiation between the energy consumptions for different effective word-widths, and therefore the effect of many optimizations that use this information will be invisible. In this paper, we show how word-width aware energy models can be generated, sensitive to effective word-width and statistical characteristics of the processed data. An example of such a word-width aware energy model for a simple Carry Lookahead Adder can be seen in Figure 2 and will be explained in Section 4. Traditional non-word-width-aware models would only provide an energy per activation for the full word-width, which corresponds to the rightmost points in the graph, and most often do not differentiate between different activities (e.g. fixed to 0.4), which leads to an even larger error.

When consecutive operations on a certain functional unit (e.g. adder) operate on data of the same word-width, which is smaller than the total width of that unit, less energy will be spent because not all circuitry is activated. When operations on data of different widths are being executed in turns, the effective active part is the maximum word-width of both. Using the corresponding value from the word-width aware energy model will then be more accurate. When the application contains different groups of operations, operating on different word-widths, there is a possibility of doing scheduling and assignment based on the word-with, to exploit this optimally.

In general width-aware models can be used in different ways. As mentioned before, they can give designers fast and fairly accurate energy estimates to steer the energy-aware mapping of an application onto a platform. In this context, as we can assume that the architecture of the platform is given, only the energy model for the specific instances of components are needed. This means e.g. only one type of adder needs to be modeled (the one that is present on the platform, e.g. the Carry Lookahead Adder model shown in Figure 2), for only one fixed total word-width. This heavily restricts the modeling effort that is required, making it a feasible and practical approach.

Another possible scenario is the usage of word-width aware models in architecture exploration, using a retargetable simulator. In this context it is the goal to optimize the architec-

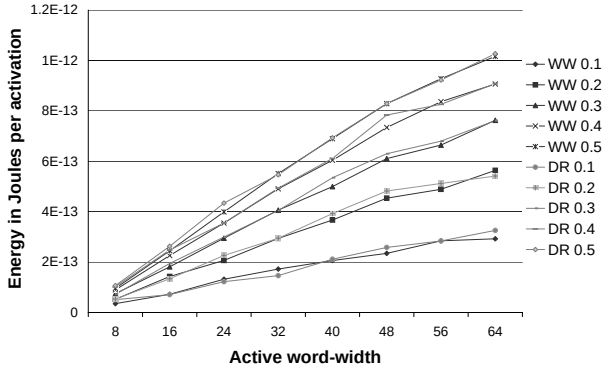

Figure 2: Word-width Aware Energy model for 64 bit Carry Lookahead SIMD Adder in 90nm CMOS standard cell technology. WW and DR indicate the Word-Width or Dynamic Range respectively, for different bit activities ranging from 0.1 to 0.5, as detailed in Section 4.2

ture components, given a set of applications or an application domain. Here models should be developed for different types of every component (eg. Carry Lookahead, Brent-Kung for adders of different widths, but also for Register Files of different Width, Depth and number of ports for Register Files, etc.), which quickly leads to a high modeling effort. This modeling should however only be done once per technology node and a big part of the flow (described in the next subsection) can be reused, making it still very worthwhile.

In the experimental section of this paper, we assume that the architecture is fixed, and present results based on energy models generated for a single adder, multiplier and register file.

2.3 Word-Width Aware Mapping

Using the extra word-width information, the result of a fixed point refinement, more optimal mappings of embedded applications can be achieved. This information can be used e.g. when deciding how to make use of SIMD to handle data parallelism. When from an algorithmic point of view the minimal precision of data is decided, this bit-width is normally promoted to the next order of two in order to fit into subwords commonly provided by embedded processors (e.g. only 8, 16 or 32 bit data). The minimal width is therefore in general not known during the final mapping. However, using this width information, more efficient mappings could be obtained using a technique called Software SIMD. This technique is illustrated below using a small artificial example, on two small pieces of pseudo code.

The original code on the left shows two loops that are operating on arrays. From fixed point refinement, we assume that the data of arrays a, b and c is found to be of minimally 18 bits, while the data of arrays d, e and f only require a 12 bit precision. On a traditional 32 bit datapath, supporting 4x8 or 2x16 bit SIMD, the loops can not be merged by using SIMD. Because of register file pressure, we assume both loops cannot be merged.

```
for i=1 to 500
    a[i]=b[i]+c[i]

for j=1 to 500
    d[j]=e[j]+f[j]
```

Original Code

```
for i=1 to 500
    Pack (b,e)
    Pack (c,f)
    (a,d)[i]=(b,e)[i]+(c,f)[i]
    Unpack (a,d) → a
    Unpack (a,d) → d
```

Transformed Code

In the right code fragment, we assume that the extra word-width information was used during mapping, and that Software SIMD was performed. By packing data of width 18 bits with other data of width 12 bits into one 32 bit word, leaving a 2 bit guard band to prevent carry overflow in between, SIMD can still be used. Hardware SIMD support is not used/required in this way and the datapath is still filled. In this case the computations of both loops have been merged and arrays `(b,e)` and `(c,f)` are operated on together, producing the combined results `(a,d)`. In this case register file pressure is reduced by combining data into less words, so the loops have been merged into one.

To be able to operate on these data together, they have to be packed and the results have to be unpacked. This requires extra operations and hence the cost of this overhead should be carefully balanced against the expected gains (for more details, see Section 6). In some cases, e.g. when the data are used together many times or when the data-layout of has been optimized, these extra operations can be avoided, which leads to different trade-offs.

Word-width aware models for the architectural components of the platform are key to be able to correctly evaluate the expected gains, and to balance this with any overhead that might be introduced by the transformation. Since Software SIMD is less restrictive toward the word-widths that can be combined, it can handle cases where traditional Hardware SIMD is not possible. Because it affects the number of accesses to memories and also the number of cycles needed to complete an algorithm, the gains depend on different parts of the platform. If the technique is applied blindly, the overhead can be larger than the gains. In this paper we show how a representative kernel from the wireless domain can be mapped in different ways onto the same platform, using word-width information, and word-width aware energy models and how this work contributes to a more accurate estimation of the expected energy gains, and therefore enables a new class of energy and performance improving optimizations by using this word-width information.

3. RELATED WORK

Most instruction set simulators and add-ons for energy estimation, like Trimaran [1], SimpleScalar [4], Wattch [7] etc. provide energy estimates based on activation count. This implies that the energy/activation does not depend on the precise data that is being operated on. Figure 2 shows that the energy of an adder does vary significantly, based on the word-width of the data that the adder is operating on and similar conclusions can be drawn for other components of the processor. It is therefore crucial to take the precise toggle activity inside the component into account

while estimating energy/power consumption of the system. Other instruction set simulators like MPARM [6], which are written in synthesizable SystemC (using a detailed modeling of all components at a low abstraction level) are capable of producing detailed results, but are extremely slow. Our previous work [11] takes into account toggle activity, but no word-width information, and does this only for the register file.

Other energy estimation methods like [10] model the energy needed for a certain operation, taking into account the previous operation, but the work is very processor specific and not-retargetable at a component level. The energy model described in [12] illustrates that the actual energy consumption depends on various factors like degree of parallelism, number of instructions and also the data-width. This is however tuned only toward the TI's TMS320C6416 processor and is not scalable to other processors. Because it is highly specialized to this specific architecture, it cannot be used for other processors or platforms or for architecture exploration.

SIMD is widely available in various processors to use the available hardware efficiently by exploiting data parallelism and to boost the performance [8, 9, 2]. *Software SIMD* is a technique that can be used to emulate SIMD processing in a processor which does not have SIMD support, as is illustrated in [5]. They strictly restrict the usage of Software SIMD to data-widths of 4x8, 2x16 bit only, as would be possible with hardware SIMD. This is a considerable limitation, given the variation in available word-widths in the embedded context, as is e.g. shown by Figure 1. Here, we do not enforce this limitation and allow different packings and even different word-widths inside one packing, which creates extra flexibility and optimization potential.

4. WORD-WIDTH AWARE MODELING

In this section we will present the word-width aware energy models that are required to steer energy optimizations that exploit word-width information. We will first introduce our view on modeling and the context of this work. Then we will give an overview of the tool-flow used for energy estimation and to build the word-width aware models. The statistical generation of input data of varying word-widths is explained and finally models for some key components are presented.

4.1 Modeling Approach

To be of practical use during mapping, energy models should be *compact*, so they can be coupled to the simulator (Instruction Set Simulator or ISS) and be used to generate fast and fairly accurate results for a realistic application size. To meet this requirement, models can be formulas or look-up table based. In this section we explain how look-up tables can be generated for all architectural components. If extra- or interpolation is required, these empirical points can be used to perform curve fitting and generate empirical formulas. Because low level (e.g. gate level) simulations, taking into account the real application data, will be extremely slow, we generate word-width aware models for statistically relevant, but randomly generated input sequences. This means that some accuracy is deliberately traded-off for simulation and design time. By profiling the particular application that is to be mapped, the statistical characteristics of the data (per array) can be collected, which allows the energy estimation to use the correct energy cost (on average) from the look-up table based energy model. This leads to fast simulation results and reasonably high accuracy. Alternatively the ISS can inspect the exact data that are being used *during simulation* and take the correct energy value from the look-up table for every operation separately, which leads to an even higher accuracy, but also will be slower. In this work, we have chosen the former approach, and combined it with our in-house ISS to generate the results shown in Section 6.

To be able to generate energy estimates for different processors and platforms, models have to be available for the considered architecture components (Adder, ALU, MUL, Register File, etc.) and parameterizable over a wide range of different instances for these components (e.g. specific type (e.g. Brent-Kung adder vs. Ripple Carry Adder, etc.) total word-width of the data, number of inputs/outputs, etc. In this paper we propose to add one extra parameter to this list, namely active *Word-Width* (WW) or *Dynamic Range* (DR) (see Section 4.2).

Fabricating and measuring hardware for all these instances is impractical due to the high number of parameters and possible combinations, but a high enough accuracy can be obtained by doing a low level simulation on a gate level design of the component. Therefore different instances of all components have been simulated using the flow shown in Figure 3 to obtain the energy estimates for statistically relevant inputs (as well as area estimates and timing verification) . This flow takes into account the layout details and detailed transistor characterization provided by the standard cell library. An optimized VHDL description for all components that have to be modeled was written and synthesized using Synopsys Design Ware and the UMC90nm libraries at a target frequency of 500 MHz. Layouts have been generated for every component and for all these designs the parasitic capacitances in the routing wires and the gate capacitances of each transistor have been extracted. The extracted netlist was then simulated in ModelSim using statistically relevant input patterns (as described in the next paragraph), covering a range of relevant word-widths. The energy per activation was finally computed from the obtained energy estimate and also the timing was used to verify if the target clock frequency can be obtained (e.g. 500 MHz). If a sufficient amount of points are simulated for a certain component, these values can then be used to perform curve fitting and to obtain empirical formulas for that component.

The final result of this modeling effort can be stored as a lookup table. Area estimates are for a unit depend on the type of unit (T, e.g. Carry Lookahead Adder vs. Brent-Kung etc.) and the Total Word Width (TWW), leading to `A=f(T,TWW)`, as is shown in Figure 3. Energy depends additionally on the activity α (likelihood that a bit changes from one word to the next) and on effective Word-Width (WW) or Dynamic Range (DR). WW and DR are explained in detail in the next section. This finally leads to `E=g(T,`α`,TWW,WW|DR)`. Energy values for all relevant combinations of these parameters are stored and the relevant values are used during the simulation to produce fast and fairly accurate energy estimates.

4.2 Varying Word-Width or Dynamic Range

To be able to bring variations in word-width or dynamic

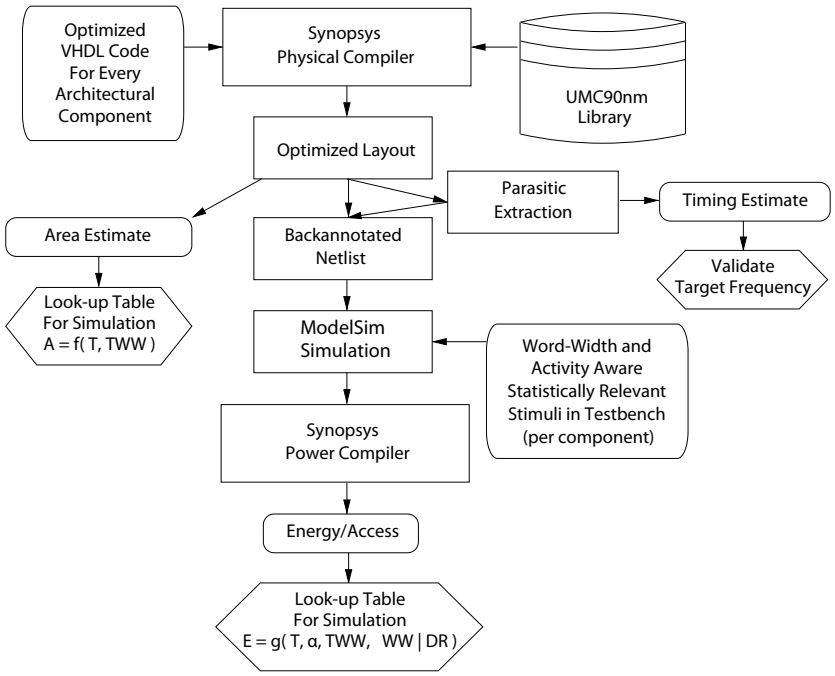

Figure 3: Word-Width Aware energy Modeling Flow

range, the input data that are simulated in ModelSim should be specified accordingly by providing different testbenches for this parameters. In our approach, we generate per component instance a set of testbenches containing automatically generated input data that have the following statistical characteristics:

- Word-Width or WW: e.g. ranging from 4 to 32 bits, in 4-bit steps (for tractability reasons): In this case we assume a component that has a Total Word-Width (TWW) of 32 bits, but of these 32 bits only the WW least significant bits are toggling. The other bits are assumed to be always 0. This case is the equivalent of operating on data of width WW of the same sign on a full 32 bit component (e.g. 8-bit addition on a 32-bit adder, when the data have been scaled with a fixed factor to be all positive [1]).

- Dynamic Range or DR: also ranging from 4 to 32 bits, in 4-bit steps: In this second case we assume that only DR bits out of the TWW are toggling, but the other bits are fixed, but not all zero. This case is the equivalent of consecutive operations on highly correlated data, as they occur in many embedded applications (e.g. video pixels of same color), leading to only a sub-part of the TWW that will be toggling.

For both cases we assume an initial distribution of 50 % 0 and 50 % 1 bits and we generate data that have an activity of 0.1, 0.2, 0.3, 0.4 or 0.5, which means that the chance that a bit is of the *active* part of the word is toggled between two consecutive is between 10 and 50 %.

[1]In this work we assume that all data has been scaled to be positive, in order to prevent all 32 bits to toggle when using 2's complement representation.

5. EXPERIMENTAL RESULTS FOR WORD-WIDTH AWARE MODELS

For the experiments shown in this paper, the flow presented in the previous section has been used to create word-width aware models for the components of a standard 64 bit SIMD enabled RISC processor, consisting of a 3-ported, 16 deep Register File with two read ports and one 1 write port (2R/1W), an adder and a multiplier. The adder supports 8x8, 4x16 or 2x32 bit SIMD additions, while the multiplier operates on half the width, respectively 8x4, 4x8 and 2x16 bit. Figure 6 shows the energy consumption per activation for multiplier. Figure 4 and Figure 5 show the energy consumption per activation for the synthesized register file, for write and read access respectively. Similar energy models have been used for load/store operations and pack/unpack operations.

These word-width aware energy models are essential when optimizing an embedded system, because they enable a more accurate estimation of the expected energy costs and gains, and therefore enable a new class of energy and performance improving optimizations using this word-width information.

In the first part of this section we will introduce the experimental setup to evaluate the added benefit of using word-width aware models. We use a fixed platform and a representative wireless benchmark. For this example, we will show the difference in accuracy when using non-word-width aware and word-width aware models respectively.

5.1 Platform and Benchmark Application

A representative wireless communication kernel is mapped onto a fixed platform, containing a small in order DSP processor and both a data (DL1) and instruction (IL1) memory. The processor has a 64 bit datapath, supporting 8, 16 or 32 bit operations (and 8x8 or 4x16 and 2x32 bit SIMD). We

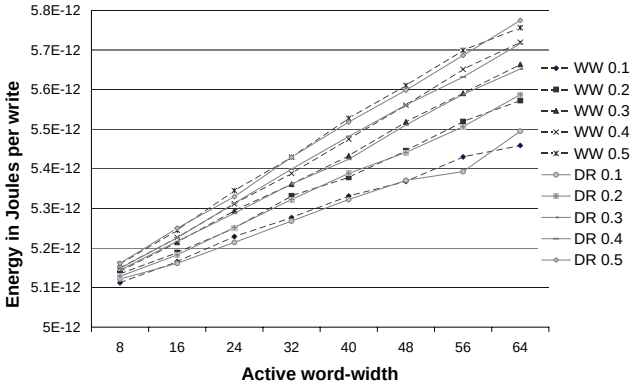

Figure 4: Energy per Write for a 1W/2R Synthesized Register File that can store 16 words of 64 bits

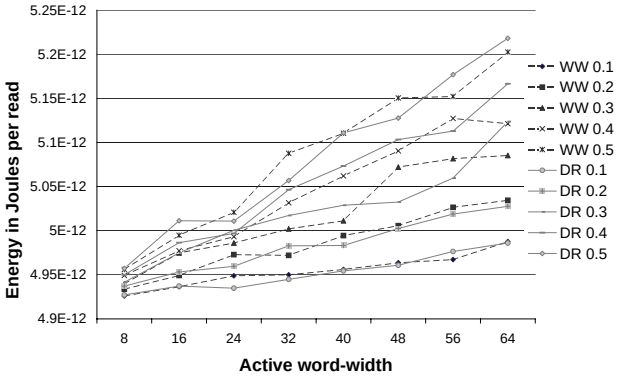

Figure 5: Energy per Read for a 1W/2R Synthesized Register File that can store 16 words of 64 bits

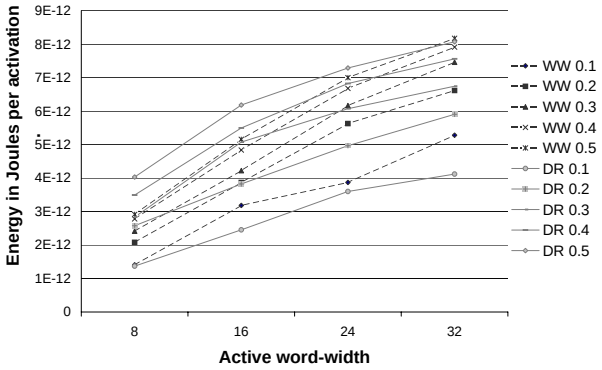

Figure 6: Energy per activation of a 32 bit multiplier (half of the width of the adder)

assume the word-width supported by the multiplier is only half the width of the other arithmetic units, as is common in current DSP processors. Intermediate data is stored in a three ported (2R/1W) register file that can store 16 words of 64 bits. This data is read from and stored back to the DL1 of 8 kB. Instructions are read directly from the IL1 instruction cache of 8 kB.

The result presented in this section are for a real life MIMO (Multiple Input Multiple Output) Baseband processing algorithm, namely the Spatial Equalizer part. This small part of the code is called for different users and different antennas in the system and is an important part of the application. The initial MIMO Spatial Equalizer code, operating on complex data, was converted to operate on the real and imaginary parts separately, as the processor does not support complex operations. The default width of the original data is 16 bits. After fixed point refinement, it is assumed here that a width of 9 bits is sufficient for the signals present in this kernel. This width depends in practice on the application requirements (e.g. BER, Bit Error Rate), and it is chosen here in the realistic range, as is shown by Figure 1).

5.2 Word-Width Aware Energy Models

To show the effect of using word-width aware energy models, we first compare an energy estimation for this kernel, using only activation based information with the improved word-width aware estimation. The *Activation Based* method, which is common in state-of-the-art ISS based energy estimation, collects the number of times a certain component is activated (e.g. a certain datapath unit or a memory) during simulation. Finally this number is multiplied with an average cost for activating that unit. In practice, this average number is a realistic estimate for using that unit at its full width (e.g. 64 bit). Figure 7 shows a comparison of *Activation Based* (no WW) and *Word-Width Aware* (WW-Aware) for the processor described above, operating on 9 bit data of the MIMO benchmark.

The plots show a severe over-estimation of the energy spent in computations on the datapath (DP) for the Activation Based method, while the Word-Width Aware method (WW-Aware) more accurately estimates the real cost when using only a small width of the available datapath effectively. For the datapath this effect is most visible (a factor 10), as a reduced active word-length heavily reduces the activity in the datapath units, like adders, shifters and multipliers (as can be seen e.g. in Figure 2, which scales almost linearly with the number of effectively active bits). Additionally we see a reduction in the energy that is spent in the register file (RF), which is computed in the same way as the datapath, using word-width aware energy models. In this case, even with the larger base cost (decoder and sense amplifiers) of the register file, the more accurate model leads to a reduction of around 10 % in the estimated energy consumption. A similar effect is expected in the Data Memory Hierarchy (DMH). Due to the unavailability of accurate models and because we cannot generate word-width aware energy models using the flow presented in Section 4, we currently use a crude estimation for the effect of word-width variations on the data memory. For state of the art embedded memory designs (DL1), roughly half of the energy is spent in the decoder and word lines, while the other half is spent on memory cells, bit lines and sense amplifiers [3]. For the second part, we expect that using smaller word-widths will reduce the energy in the bit lines about linearly, which leads to a reduction of about 20 % in this specific example. Since the

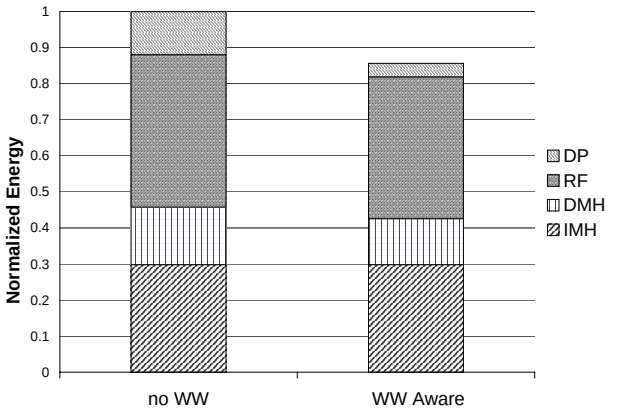

Figure 7: Comparison between activation based energy estimation (*no WW*) and energy estimation based on activation information using word-width aware models (*WW Aware*).

Instruction Memory Hierarchy (IMH) is not directly linked to the effective word-width of the data, there are no gains in this part. This part will still be somewhat affected, but since the number of instruction bits corresponding to the datapath is relatively small compared to the total number, the effect on the cost per activation will be small and it is ignored here. We do however consider a change in the number of accesses due to different mapping, and the corresponding effect on the energy consumption. Fore wider processors (e.g VLIW, Very Long Instruction Word processors) extra accuracy could be needed. In this case the instruction encoding should be fixed first and a more detailed energy model, depending on the effective instruction should be used. This is outside the scope of this paper.

Figure 7 shows an overall difference of about 15 % between the non-word-width aware and the word-width aware method, in the energy estimation of the kernel for the full platform. This is a quite significant difference and motivates the development of word-width aware models. Not doing so, can lead to wrong conclusions, or wasted effort in optimizing the platform. Based on the *no WW* plot, one might decide to spend effort on techniques that try to reduce the energy spent in the datapath, even at the cost of a small overhead. In reality the cost of the datapath is much lower and the small overhead could introduce an overall energy penalty. It should also be noted the exact difference depend on the chosen processor and platform. For some architectures, like coarse-grained or vector processors, the relative contribution of the datapath can be much larger than shown in this plot, leading to even larger differences.

6. SOFTWARE SIMD

In this section, we show for one example optimization that significant gains in both energy and performance can be obtained during the mapping process when word-width information is exploited. We explain the different possibilities in mapping, exploiting either SIMD supported by hardware (SIMD in the traditional sense, here referred to as Hardware SIMD) or Software SIMD or using no SIMD at all. For the SIMD cases, we also look at the way data is stored initially in memory. We then show the energy and performance estimations for each of these mappings and draw some conclusions.

6.1 Hardware SIMD vs Software SIMD

Current energy efficient and high performance processors use Hardware SIMD (Single Instruction Multiple Data) to make use of the inherent data parallelism of applications and to obtain a higher energy efficiency and to boost performance. This approach operates in parallel on multiple data words of the same word-length on a wider datapath. SIMD processors provide special hardware in their datapaths that supports computations on full words and on a limited set of sub-words of the same length (e.g. 2x32 bits or 4x16 bits or 8x8 bits), producing multiple independent results. Standard 32bit processors use Hardware SIMD to use their hardware up to its full potential when working on e.g. 8bit video data, operating on four independent 8bit words together. In many cases however, the limited number of word-lengths that are supported can restrict the mapping freedom, because data always needs to fit into the provided sub-word lengths. When the smallest sub-word supported by the hardware is e.g. 8 bit, an operation on 4 bit data has to be promoted to 8 bit sub-words, which is not really efficient. By providing more detailed word-width info to the mapping process, different and potentially more efficient combinations in mapping can be found. These detailed word-widths should not be restricted to pre-defined word-widths or rounded to the next subword size (of higher width) that is supported by hardware, as is traditionally done.

A so called *Software SIMD* approach can combine different word-widths (e.g. 12 bits + 18 bits on a 32 bit datapath, as was shown in Section 2) that would not be supported by the hardware, or could even be used to make use of SIMD on a datapath that does not support this in hardware (no hardware support for carry chain splitting). If the data that are merged into a single operation are used together for a high number of times during the computations, the overhead of packing, shifting and unpacking them can be limited. Modern applications, especially in the wireless context, feature this kind of behavior.

By using word-width information to group data together and perform Software SIMD, the energy consumption of many different parts of the platform are directly affected. As with Hardware SIMD, the number of operations is reduced, leading to less accesses to the instruction memory hierarchy. If different shorter data words are stored together in a single word, the number of accesses to the data hierarchy is reduced. As also less operations have to be scheduled, a higher performance can be possible, leading to an increase in performance.

Before different data can be operated on together, they have to be packed into a single word. After all computation on that combination have been performed, the independent data have to be recovered by the inverse unpack operation. If these data are used together for a sufficient number of times, the extra overhead of the pack/unpack operations which includes the accesses to the instruction and data memory linked to these operations will be small compared to the gains of operating on them together. However, if this is not the case, these extra operations and accesses are in fact leading to a negative gain.

In addition to packing different sub-words of equal and pre-defined lengths (as supported by the hardware) into one word, as is done in traditional SIMD, in this work we will not restrict packing to words of equal length, therefore creating freedom and optimization potential in applications that cannot use Hardware SIMD. Even in mappings that already make use of traditional SIMD, a more efficient mapping could be obtained when making use of the more flexible Software SIMD. This extra flexibility, leading to many more potential SIMD combinations, however leads to an increased complexity in estimating the effect of applying the Software SIMD. The estimation has to include the effect of the extra pack/unpack operations that can be more complex in order to fix the alignment of the subwords in a more flexible way than is needed to align to hardware boundaries only (a combined shuffle and shift operation will be needed to transform the original data to the appropriate data layout).

It should be noted here that Figure 1 shows the minimum number of bits that is needed to represent the maximum signal values without overflow, without extra saturation logic to prevent wrap around. For signals that require saturation logic, because of instable behavior resulting from feedback in the system (e.g. Infinite Impulse Response filters in audio), this technique can not be used.

6.2 Using Word-Width Info for Mapping

In this section we explore the potential gains of exploiting word-width information by presenting five different mappings:

1. No SIMD: In this case we assume that no SIMD is used on the processor (as a pessimistic baseline case), and all data are aligned in the data memory to 64 bit boundaries. On the processor the 32 bit operation mode is chosen, keeping the 32 MSB bits fixed to 0.

2. Hardware SIMD: Our second reference case assumes that the Hardware SIMD provided by the datapath hardware is exploited and all data are aligned in memory to 16 bit boundaries (so in each subword, that actually contains only 9 bit data, the 7 MSB bits are 0). In this case the data are pre-packed when they are read from the memory in an optimal way for the datapath (4x16 bit mode) and also can be stored back packed.

3. Hardware SIMD with Pack/Unpack: If the data is not pre-packed in memory, the Hardware SIMD capability of the datapath can still be exploited, but extra pack and unpack operations have to be performed in order to transform the data layout. In this case the datapath operates in the 4x16 bit mode, but the data layout is assumed to be aligned to 64 bit boundaries (only 9 bits out of 64 contain the actual data). To transform the data layout we combine four loaded words into one word, using a pack unit, supporting a combined shuffle/shift. At the end of the computations the words are unpacked and stored back using the original data layout. This introduces in 3 pack and 4 unpack operations per SIMD operation.

4. Software SIMD: By explicitly exploiting the knowledge from fixed point refinement, a more efficient mapping can be obtained by directly operating on the 9 bit subwords. We can operate on 6 subwords of 9 bits in parallel, filling the 64 bit datapath almost completely. In this case we assume that data are pre-packed in memory according to 10 bit boundaries (9bit data + 1 guard band bit), which is realistic if the loaded data are consumed in this packed fashion for a high number of times and in different places of the algorithm. If this is the case, we only have to pack/unpack them once, at the beginning and at the very end, thereby reducing the overhead of these extra operations resulting from transformations in the data layout. The hardware is activated in the 2x32 bit mode, and the data layout respects the 32 bit boundary in the middle.

5. Software SIMD with Pack/Unpack: If the fixed point refined data are not pre-packed, or if they are only used in the specific packing that is needed in this kernel once, we cannot neglect the packing/unpacking overhead. In this case we include the extra (shuffle/shift) operations needed to transform an initial data layout of 64 bits to the packed version of the previous case. This introduces 5 pack operations and 6 unpack operations for every Software SIMD enabled operation.

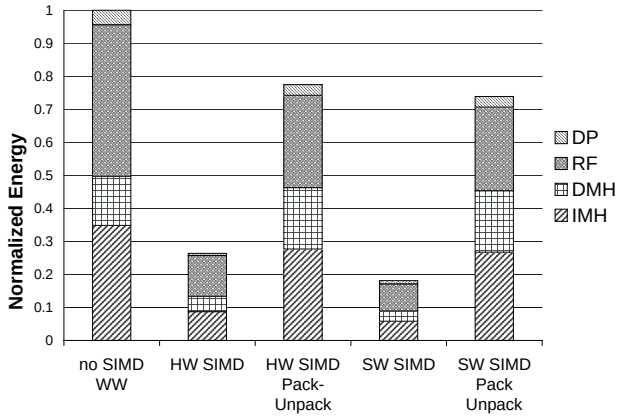

Figure 8: Energy Breakdown of different mapping variants of the MIMO kernel, using different SIMD options and data layouts

Figure 8 and 9 show the energy breakdown and performance of all mapping variations described above, using word-width aware models in all cases (for the energy plot). The results have been normalized to the energy consumption and performance of case 1, which serves as a pessimistic baseline case. For case 1, no SIMD is used on a 64 bit wide datapath with SIMD support, which leads to a high energy cost and a bad performance (as is to be expected). In this case every data item that is needed during the computation is loaded from the memory to the register file, operated on and eventually stored back individually.

Using the Hardware SIMD capabilities provided by the hardware, and assuming a SIMD optimized data layout, the energy consumption is reduced by more than 70 % (see Case 2, Hardware SIMD) and performance is improved with 75 %. This is the result that can be achieved by a state of the art approach, using a good mapping. In this case the number of loads from the memory to the register file is heavily reduced

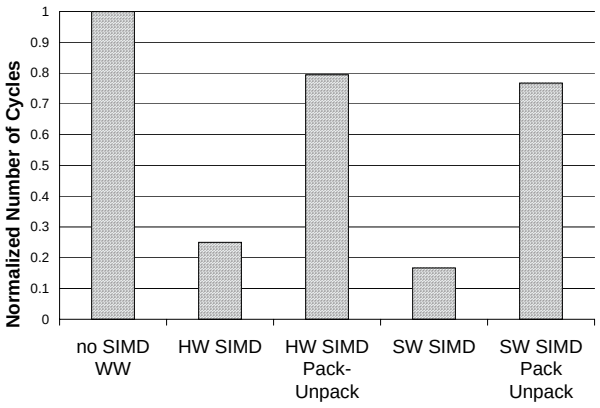

Figure 9: Performance results of different mapping variants of the MIMO kernel, using different SIMD options and data layouts

by loading multiple data in parallel (4 data items of 9 bits, each promoted to 16 bits in the full 64 bit datapath width), which leads to energy savings in both the register file and the memory. Also the number of computational operations is reduced by the same factor, which leads to less accesses to the instruction memory.

By exploiting extra word-width information and performing Software SIMD, we can reduce the energy even more. When, as in the previous case, we assume that the data layout has been optimized (see Case 4, Software SIMD), we can gain over 80 % compared to the baseline or over 30 % compared to Case 2. Performance is improved with 83 % compared to Case 1 and with over 33 % to Case 2. In this case more data items are packed together in one word (6 data items of 9 bits, each with a 1 bit guard band, filling the 64 bit datapath width), which lead to even less accesses to the data memory and the register file and to less operations and accesses to the instruction memory hierarchy.

When the data-layout has not been optimized, or when different packings are needed in different parts of the application, the overhead of the pack and unpack operations cannot be removed completely or neglected. When transforming the data layout in order to use Hardware SIMD (see Case 3, Hardware SIMD with Pack-Unpack) the number of datapath operations that is performed on the packed data is still smaller than in Case 1, but a high number of pack and unpack operations are added to make this possible, which leads to a larger energy cost in the datapath itself. The number of accesses to the register file are however overall still reduced. The number of accesses to the data memory and the energy cost of this part are exactly the same, as the original and final data layouts are the same. Compared to Case 1 the energy consumption is still reduced with over 20 % and performance is improved with about the same amount.

By adding extra word-width information and performing Software SIMD under the same constraints as Case 3, we can still gain an additional 3.5 % compared to Case 3, on both energy and performance. In this case the number of accesses to the register file and the number of operations is still marginally reduced, but even more pack and unpack operations had to be added, which leads to an overall gain of just a couple percent. However, this is still a gain on the processor, including the data and instruction memories (DL1 + IL1).

It should be noted here that without going through the effort of modeling the effective word-width to estimate the energy, gains would look very different here. Without word-width aware energy models the energy cost of the datapath would be severely overestimated in Case 1, as is shown in Figure 7, and less in the other mappings, as there the datapath is much better filled. This would lead to huge expected gains, which in reality would not be present. In some cases this would justify more expensive data layout transformations, which in the end might lead to an overall energy loss.

7. CONCLUSION

In this paper we show the value of exploiting word-width information of application data during application mapping or optimization. To correctly estimate the impact of these optimizations on the energy consumption or on the performance, we introduce word-width aware energy models for key platform components and show how the improved accuracy of these models is essential to steer the optimizations (20 % difference with non-word-width-aware conventional approach). Word-width aware energy models can be generated using the presented flow, and the models can be coupled to existing Instruction Set Simulators to generate fast and accurate estimates. Using a representative benchmark from the wireless communication domain, we show that significant gains can be obtained when exploiting word-width information during mapping, using a technique called Software SIMD as an illustrative example optimization. A reduction in energy consumption and an improvement in performance of about 80 %, when compared to using no SIMD, or of over 30 % when comparing to hardware SIMD, was demonstrated for this example (using the proposed word-width aware energy models). Traditional energy estimation, using non-word-width-aware models would heavily overestimate the gains, which would lead to incorrect optimization decisions in some cases.

8. FUTURE WORK

In the future compilers can be extended to accept the word-width information from the designer, or to extract it through profiling or in an other way, in order to automatically produce more optimal mappings. In order to make this automation possible, many trade-offs have to be modeled and evaluated. Currently we are extending our ISS and automated energy estimation flow in order to make these trade-offs visible. We are evaluating how to remove the overhead of input data scaling and how to handle sign extensions for signed data. Finally, we are looking at how to exploit word-width information also during scheduling and assignment.

9. ACKNOWLEDGMENTS

This work has been supported by the IWT Flanders.

10. REFERENCES

[1] *Trimaran: An Infrastructure for Research in Instruction-Level Parallelism.* http://www.trimaran.org, 1999.

[2] S. Agarwala, P. Koeppen, T. Anderson, A. Hill, M. Ales, R. Damodaran, L. Nardini, P. Wiley, S. Mullinnix, J. Leach, A. Lell, M. Gill, J. Golston, D. Hoyle, A. Rajagopal, A. Chachad, Agarwala, R. Castille, N. Common, J. Apostol, H. Mahmood, M. Krishnan, D. Bui, Q.-D. An, P. Groves, L. Nguyen, N. Nagaraj, and R. Simar. A 600 mhz vliw dsp. In *Solid-State Circuits Conference*, 2002.

[3] B. Amrutur and M. Horowitz. Speed and power scaling of SRAM's. In *IEEE Journal of Solid-State Circuits*, volume 35, February 2000.

[4] E. D. Austin T., Larson E. Simplescalar: an infrastructure for computer system modeling. *IEE Computer Magazine*, 35(2):59–67, 2002.

[5] BDTI, http://www.insidedsp.com/tabid/64/articleType/ArticleView/articleId/173/Implementing-SIMD-in-Software.aspx. *Implementing SIMD in Software*, June 2006.

[6] L. Benini, D. Bertozzi, A. Bogliolo, F. Menchelli, and M. Oliviri. Mparm: Exploring the mulit-processor soc design space with systemc. In *Journal of VLSI Signal Processing*, volume 41(2), pages 169–182, 2005.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc of the 27th International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[8] P. Hofstee. Power efficient processor architecture and the cell processor. *High-Performance Computer Architecture, HPCA-11*, pages 258–262, 2005.

[9] Intel, http://www.intel.com/support/processors/sb/cs-001650.htm. *Streaming SIMD Extension 2 (SSE2)*.

[10] N. Julien, J. Laurent, E. Senn, and E. Martin. Power consumption modeling and characterization of the ti c6201. In *IEEE Micro*, Sep-Oct 2003.

[11] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, and D.Verkest. EMPIRE: Empirical power/area/timing models for register files. In *International Journal on Embedded Systems (special issue on Media and Stream Processing)*, 2006. To appear.

[12] M. Schneider, H. Blume, and T. G. Noll. Power estimation on functional level for programmable processors. *Advances in Radio Science*, 2:215–219, May 2004.

[13] C. Shi and R. W. Brodersen. Automated fixed-point data-type optimization tool for signal processing and communication systems. *Proceedings of the Design Automation Conference*, pages 478–483, 2004.

A Coprocessor Accelerator for Model Predictive Control

Panagiotis Vouzis & Mark Arnold
Computer Science and Engineering Dept.
Lehigh University
Bethlehem, PA 18015, USA
{vouzis, maab}@lehigh.edu

Leonidas Bleris
Bauer Laboratory
Harvard University
Cambridge, MA 02138, USA
lbleris@cgr.harvard.edu

Mayuresh Kothare
Chemical Engineering Dept.
Lehigh University
Bethlehem, PA 18015, USA
mayuresh.kothare@lehigh.edu

Yongho Cha
Advanced Digital Chips
Seoul, 135-270, Korea
cygx1@adc.co.kr

Model Predictive Control (MPC) [1] is an established control technique that has found widespread application mainly in the chemical-process industry. There is increased interest for its introduction into a wide range of nonindustrial applications due to its ability to handle Multiple-Input-Multiple-Output (MIMO) systems and to take into account constraints and disturbances explicitly. Notably, the explicit handling of constraints by MPC makes it the algorithm of choice for safety-critical control systems found in small physical size applications, such as drug delivery [2], and microchemical systems that encompass a chemical system with a number of actuators and sensors [3]. The efficient implementation of a Digital-Signal-Processing (DSP) system for MPC requires a custom-designed architecture that is tailored to the particular algorithm in order to exhibit reduced area and consequently reduced power consumption, while meeting the real-time operation requirements.

The limiting factor for the adoption of MPC by small physical size applications is its high computational requirements. On each time-step MPC requires the solution of an optimization problem. When MPC is used for industrial applications, which are systems with slow dynamics, a general-purpose high-end workstation can be utilized to carry out the necessary computations. For applications, whose physical size is small, the limited area and power constraints have to be addressed carefully by a custom-designed hardware architecture.

In this work we propose an architecture that consists of a general-purpose microprocessor and an auxiliary unit, tailored to accelerate computationally-demanding MPC operations. For example, the basic update step of the Newton optimization algorithm is given by the equation

$$u(t+1) = u(t) - H(f(u))^{-1} \cdot \nabla(f(u)), \qquad (1)$$

where $u(t+1)$ is the new control move, $u(t)$ the previous move, $H(f(u))^{-1}$ is the inverse of the Hessian of the objective function, $f(u)$, and $\nabla(f(u))$ the Gradient of the objective function. We can see there are matrix inversions, matrix-by-vector multiplications, vector subtractions and also a number of matrix operations required by the Hessian and Gradient

The design objective for implementing MPC is an architecture that occupies small area, is efficient enough to meet the real-time requirements of the target application, and it can be combined with a general-purpose microprocessor, such as the ones encountered in embedded systems. Towards this direction, a co-design methodology is used to develop an architecture that is efficient both in power consumption and performance, while it is sufficiently flexible to be embedded in bigger systems that need to include MPC in their functionality. Co-design combines softwares's flexibility, which is executed on the general-purpose microprocessor, with the high performance offered by hardware.

The computationally intensive parts of the algorithm are implemented in hardware by a matrix coprocessor in order to reduce the computational delay and free the microprocessor from the burden, while software is used to carry out algorithmic-control tasks and high-level operations. The total number of arithmetic operations is a substantial load for an embedded general-purpose microprocessor which may have to carry out a number of other tasks required by the embedded application. The decision for the partitioning of the MPC between software (microprocessor) and hardware (coprocessor) is the most critical one in the whole process, and it is based on a profiling study of the algorithm which helps to identify its bottlenecks. The analysis led to the transfer of the calculation of the Gradient, the Hessian and the inverse Hessian to the coprocessor, while the rest of the algorithm is executed on the general-purpose microprocessor.

The path of the co-design process begins with the setting of the specifications, and the software-hardware partitioning follows. After the communication protocol between the two parts is specified, these are implemented by using a Hardware Description Algorithm (HDL) for the hardware and a high-level programming language for the software. The next step is to co-simulate the two parts in order to verify the correct functionality and the performance of the complete design. If the verification process fails, then a backward jump is made to the appropriate design step, e.g., if the performance of the system does not meet the specifications, then a new partitioning decision is made and the design process is repeated.

The matrix coprocessor encompasses a one-hot Finite State Machine (FSM) and an Arithmetic Logic Unit (ALU) that can carry out one multiply-accumulate operation per clock cycle, i.e., it operates in an iterative fashion on the matrix elements. The implementation of the FSM was developed by using a tool called VITO, which is a preprocessor for the Verilog HDL.

VITO simplifies the description of one-hot FSMs to software-like statements (while loops, if-then-else branches, etc.) which speed-up the development process [4].

The software part is developed in the C programming language which has an extended instruction set consisting of the commands that are executed by the matrix coprocessor. There are conventional commands, such as loading a matrix (LOADC), storing a matrix (STOREC), outputting a matrix (OUTC), and novel custom-designed ones such as calculating $1/a_i^2$ and $1/a_i^3$ for a vector a (POW2A, POW3A), multiplying a matrix $\mathbf{C}$ by vector b (MULV), finding the pivot line for Gauss-Jordan inversion (PIVOT). The general-purpose microprocessor acts as the master in the system; i.e., it carries out the tasks of Input/Output (I/O), initializes and sends the appropriate commands to the auxiliary unit and receives back the optimal control moves. The auxiliary unit acts as a matrix coprocessor by carrying out matrix operations, such as addition, multiplication, etc. While the coprocessor executes a command, the microprocessor can run any other task.

The wide dynamic range required by MPC does not favor the use of a fixed-point number system, thus the viable alternatives are between the Floating-Point (FP) number system and the Logarithmic Number System (LNS) [5], which has been adopted for the arithmetic operations on the coprocessor. The utilization of LNS allows the reduction of the required word-length to 16 bits, and consequently a general-purpose microprocessor of the same word-length is used. The alternative of an equivalent FP unit, although exhibiting similar delay as the LNS, is proven to occupy 40% more area [6], and thus consumes more power. The adoption of LNS in this work, instead of FP, is based on the study by Garcia et al. [6] where it is shown that a reduced-precision LNS is capable of solving an MPC problem efficiently in terms of performance and area, which are very critical aspects when low-power consumption and embeddability are sought. The same authors in [7] propose an LNS-based Application-Specific-Instruction Processor (ASIP) of reduced precision suitable for MPC algorithms. Further proof of the applicability of LNS for MPC problems is given in [8]. This work presents a different approach in the utilization of LNS in terms of architecture, since we propose a design that consists of two parts, the microprocessor and the coprocessor that incorporates an LNS unit with a algorithm-specific instruction set, and not a single ASIP that uses LNS to carry out the arithmetic operations as in [7].

Both the microprocessor and the coprocessor are described in Verilog and were simulated by using ModelSim XEIII/Starter 6.0a in order to verify the functionality and measure the performance of the system. For implementation purposes we selected the 16-bit Extensible Instruction Set Computer (EISC) of ADCUS to act as the host. The target technology for synthesis is a Field Programmable Gate Array (FPGA) of Xilinx; thus the development environment ISE 7.1 of the same vendor is used. The program running on the microprocessor is developed using the EISC Studio of ADCUS. A prototype is developed using the Xilinx ML401 board which hosts a XC4VLX25-FF668-10C Virtex-IV FPGA.

The functionality of the system was tested with hardware-in-the-loop simulation for two control problems—a linear antenna-control problem [9] and the nonlinear glucose-regulation problem for diabetic patients [10]. The antenna-control problem can be solved by using Motorola's 32-bit MPC 555 processor, running at 40MHz and incorporating a 64-bit FP unit (double precision), in 15ms [11], while the microprocessor-coprocessor proposed architecture can solve the same problem in 0.89ms running at 5MHz. We can see, for the particular problem, there is a 27 times speed-up (nomalized to the same clock speed) while the 64-bit FP unit of the Motorola MPC 555 occupies 17 times more area than the 16-bit LNS arithmetic unit [7]. This is an example of the amount of speed-up and area reduction that can be achieved by using the custom-designed microprocessor-coprocessor architecture, instead of an off-the-shelf general-purpose microprocessor.

The microprocessor of an embedded systems should be chosen to be the most economical one in terms of performance, wordlength, peripherals, and memory size—all of which aim to a system with reduced power consumption and cost. When such an embedded system needs to incorporate MPC functionality, the proposed matrix coprocessor offers a cost-efficient solution capable of bearing the computational effort of the algorithm in real time. The interested reader is referred to [12] for more details on the architecture and the design process.

REFERENCES

[1] E. F. Camacho and C. Bordons, *Model Predictive Control*. Springer-Verlag, London, 2nd edition, 2004.

[2] R. S. Parker, F. J. Doyle III, and N. A. Peppas, "A Model-Based Algorithm for Blood Glucose Control in Type I Diabetic Patients," *IEEE Trans. on Biomedical Engineering*, vol. 46, pp. 148–156, Feb. 1999.

[3] K. F. Jensen, "Microchemical Systems: Status, Challenges, and Opportunities," *AIChE Journal*, vol. 45, pp. 2051–2054, Oct. 1999.

[4] M. G. Arnold and J. Shuler, "A Processor that Converts Implicit Style Verilog into One-hot Designs," in *Proceedings of the* 6^{th} *International Verilog HDL Conference*, (Santa Clara, CA), pp. 38–45, 1997. www.verilog.vito.com.

[5] E. E. Swartzlander and A. G. Alexopoulos, "The Sign/Logarithm Number System," *IEEE Trans. on Comp.*, vol. 24, pp. 1238–1242, Dec. 1975.

[6] J. G. Garcia, M. G. Arnold, L. G. Bleris, and M. V. Kothare, "LNS Architectures for Embedded Model Predictive Control Processors," in *Proceedings of the 2004 CASES International Conference*, (Washington, DC), pp. 79–84, Sept. 2004.

[7] L. G. Bleris, J. G. Garcia, M. V. Kothare, and M. G. Arnold, "Towards Embedded Model Predictive Control for System-on-a-Chip Applications," *Journal of Process Control*, vol. 16, pp. 255–264, March 2006.

[8] L. G. Bleris, J. G. Garcia, M. G. Arnold, and M. V. Kothare, "Model Predictive Hydrodynamic Regulation of Microflows," *Journal of Micromechanics and Microengineering*, vol. 16, pp. 1792–1799, Sept.2006.

[9] M. V. Kothare, V. Balakrishnan, and M. Morari, "Robust Constrained Model Predictive Control Using Linear Matrix Inequalities," *Automatica*, vol. 32, pp. 1361–1379, Oct. 1996.

[10] R. S. Parker, F. J. Doyle III, and N. A. Peppas, "The Intravenous Route to Blood Glucose Control," *IEEE Engineering in Medicine and Biology*, vol. 20, pp. 65–73, Jan./Feb. 2001.

[11] L. G. Bleris and M. V. Kothare, "Real-Time Implementation of Model Predictive Control," in *Proceedings of the American Control Conference*, (Portland, OR), pp. 4166–4171, June 2005.

[12] P. Vouzis, L. Bleris, M. Kothare, and M. Arnold, "A System-on-a-Chip Implementation for Embedded Real-Time Model Predictive Control," submitted to *IEEE Trans. on Control Systems Technology*, 2006.