2014 IEEE 28th International Parallel & Distributed Processing Symposium

Using Multiple Threads to Accelerate Single Thread Performance

Zehra Sura, Kevin O'Brien, and Jose Brunheroto

IBM T.J. Watson Research Center
Yorktown Heights, New York, USA

{zsura,caomhin,brunhe}@us.ibm.com

***Abstract*—Computing systems are being designed with an increasing number of hardware cores. To effectively use these cores, applications need to maximize the amount of parallel processing and minimize the time spent in sequential execution. In this work, we aim to exploit fine-grained parallelism beyond the parallelism already encoded in an application. We define an execution model using a primary core and some number of secondary cores that collaborate to speed up the execution of sequential code regions. This execution model relies on cores that are physically close to each other and have fast communication paths between them. For this purpose, we introduce dedicated hardware queues for low-latency transfer of values between cores, and define special *enque* and *deque* instructions to use the queues. Further, we develop compiler analyses and transformations to automatically derive fine-grained parallel code from sequential code regions.**

We implemented this model for exploiting fine-grained parallelization in the IBM XL compiler framework and in a simulator for the Blue Gene/Q system. We also studied the Sequoia benchmarks to determine code sections where our techniques are applicable. We evaluated our work using these code sections, and observed an average speedup of 1.32 on 2 cores, and an average speedup of 2.05 on 4 cores. Since these code sections are otherwise sequentially executed, we conclude that our approach is useful for accelerating single thread performance.

I. Introduction

Systems that scale out to many thousands of processing cores provide powerful computational resources and the potential for very fast execution of applications. However, this potential is often not fully realized, since it can be challenging to effectively use multiple cores to speed up the execution of a single application.

We can speed up an application by distributing its computation across multiple cores. This requires identifying parts of the application code that can execute in parallel. Parallelism can be specified in the source code using an explicitly parallel programming language or library such as Java, POSIX threads[1], MPI[2] or OpenMP[3]. Programmer-specified parallelism is useful since it can take advantage of the semantics of a particular application or algorithm. However, this parallelism is usually coarse-grained, and it is tedious and error-prone to explicitly program fine-grained parallelism. When scaling to many thousands of cores, it is important to exploit maximum available parallelism, and therefore the need to automatically detect fine-grained parallelism, even within previously identified coarse-grained parallel regions.

Fine-grained parallelism can be automatically detected in hardware, or in system software. Hardware techniques such as out-of-order issue and speculative execution enable very fine-grained instruction level parallelism. These techniques require no input from the user, and since they apply to dynamic execution, they can exploit parallelism that is specific to a single instance of running the application. However, previous studies have shown that for regular parallel codes, using multiple lightweight simple cores results in greater power-performance efficiency compared to using a single heavy-weight out-of-order core[4]. Therefore, it is preferable to use software to detect fine-grained parallelism instead of building more sophisticated but even more power-hungry hardware.

Our approach is to use the compiler to automatically detect and implement fine-grained parallelism. Our technique applies to execution environments with more hardware cores or threads than can be used by the parallelism encoded in the code or exposed using conventional compiler transformations. We define an execution model where these extra hardware cores (or threads) are designated to be *secondary* cores (or threads). One or more secondary cores (or threads) are associated with a specific *primary* core (or thread), and they work collaboratively to accelerate the execution of serial code on the primary core (or thread).

In our system, the compiler statically determines how serial code is to be distributed into multiple threads of execution, and transforms the code into a parallel version. The compiler has the advantage of being able to extract parallelism from large code regions instead of being limited to some window of dynamic instructions. Further, it is possible for the compiler to integrate high-level semantic information in its analysis. Figure 1 shows a simplified example to illustrate the fine-grained parallelism available in a code snippet, and how this computation can be distributed over two cores. In the example, variables a, b, c, d, e, x, y, and z refer to shared memory locations, and t1-t5 refer to temporaries within each core. The code snippet has 3 multiply and 2 add operations, which are distributed between two parallel threads. The fine-grained parallelism shown relies on the ability of the cores to efficiently communicate values to one another. This communication is denoted in the code by SEND(value, core-id) and RECV(core-id) functions, where core-id identifies a specific core to send the value to, or receive a value from. In practice, communication may be implemented

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.104

using shared memory. However, in our work, we consider the use of simple hardware support for low latency core-to-core transfers. This is because the communication delay affects execution time, and limits the granularity of parallelism that is beneficial for performance.

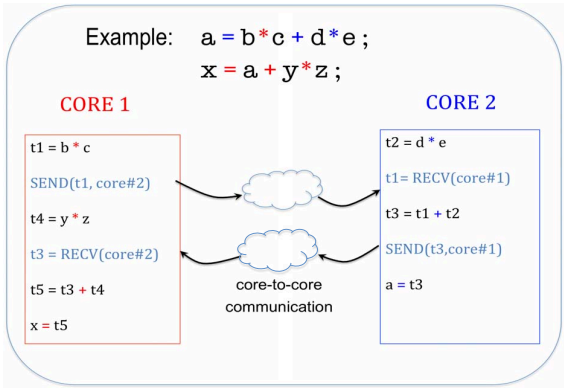

Fig. 1. An Example of Code Transformed for Fine-Grained Parallelism

Figure 2 shows another example of code transformed for fine-grained parallelism. This code is a loop taken from the Sequoia benchmark *lammps*[5], and the loop body is transformed to execute in a pipelined fashion across 3 cores, using 4 pairs of SEND-RECV operations to communicate between cores.

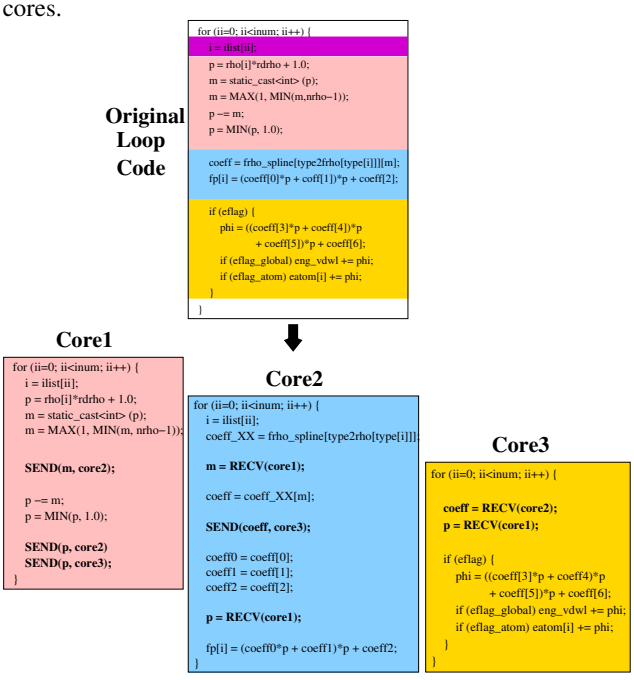

Fig. 2. Pipelined Execution for an Example Loop From *lammps*

This paper is organized as follows: we describe our execution model, together with the specialized hardware support for low latency core-to-core transfers in Section II; we give details of the compiler analysis and transformations used to extract fine-grained parallelism in Section III; we characterize code in the Sequoia tier 1 benchmarks and discuss how we choose the code sections that are the focus of our experiments in Section IV; and we present the experimental analysis performed and the results obtained in Section V. In Section VI, we discuss related work, and finally, in Section VII, we present our conclusions.

II. Hardware Execution Model

In a large-scale shared-memory processor system, hardware resources for parallelism are available at different hierarchical levels: multiple hardware threads at the core level, multiple cores at the chip level, multiple chips at the node level, and multiple nodes at the system level. In this work, the fine-grained parallelism is primarily targeted for execution on multiple hardware cores. Our technique can also be applied to multiple hardware threads on the same core, but we have not experimented with this option yet. We aim to parallelize code that is either sequential, or code that despite having been parallelized at a coarse-grain, still does not utilize all available hardware cores.

Given a serial code section, there is one *primary* core on which it executes, and a number of *secondary* cores associated with the primary core that may be used to speed up the execution on the primary core. All the associated cores are physically close to each other, so they are able to quickly communicate with one another.

Communication between cores on a chip occurs through shared memory, typically at the L2 cache level, with latency in the order of tens of cycles, and even larger latencies if it misses the L2 cache. However, since the performance of fine-grained parallelism is very sensitive to communication delays, we experiment with simple hardware support for fast core-to-core transfers. We introduce fixed-size communication queues between cores, as illustrated in Figure 3. Besides fast communication, queues have the advantage of also providing more predictable transfer times. Conceptually, for every pair of cores A and B, there is a queue dedicated to transfers from A to B, and another queue dedicated to transfers from B to A. New instructions are added to insert/remove values into/from these queues. These instructions are as follows:

- *enque*: used to send a value to another core. It takes a queue identifier and a register as parameters. The queue identifier determines the destination core, and the value in the register is placed in the next available slot in the corresponding queue. If there is no empty slot, the instruction execution stalls until a slot becomes available.
- *deque*: used to receive a value from another core. It takes a queue identifier and a register as parameters. The queue identifier determines the source core, and the next available value in the corresponding queue is loaded into the register. If there is no valid entry in the queue, the instruction execution stalls until one becomes available.

We base our evaluation (described in Section V) on a Blue Gene/Q system simulator[6][7], and include the newly intro-

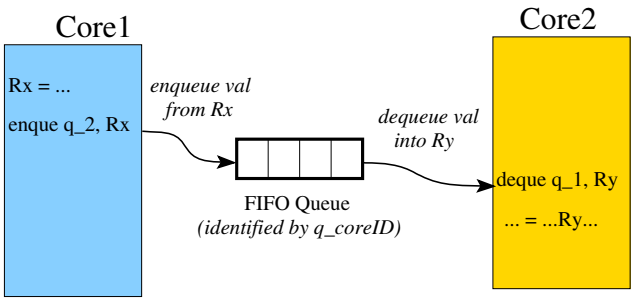

Fig. 3. Hardware Queues for Fast Core-to-Core Communications

duced *enque* and *deque* instructions in the simulation of the Blue Gene/Q chip. We choose to base our evaluation on a Blue Gene/Q system because it exemplifies the large-scale systems we are targeting in our work. The Blue Gene/Q chip contains processing cores called A2 cores, which are highly pipelined and support 2-way concurrent instruction decode and issue. The A2 cores are optimized for throughput performance, and include high-accuracy dynamic branch prediction. However, they are in-order for all of instruction dispatch, execution, and completion. This makes them power efficient, but also limits the amount of instruction-level parallelism within a single thread of execution.

The runtime performance is greatly impacted by the transfer latency when using queues for core-to-core communication. In Section V, we present data for latencies of 5, 20 and 50 processor cycles. The 5-cycle estimate is an aggressive/optimistic estimate, using the 6-cycle L1 latency for A2 cores as a base number, and assuming that special dedicated hardware for communication queues can better or match this number. Using a 5-cycle latency enables us to study the performance potential for our fine-grained parallelism technique.

With special dedicated hardware for core-to-core communication queues, we do not expect to scale to a large number of cores. This is because the model assumes all-to-all communication with individual queues for each pair of cores, and the number of queues grows exponentially with the number of cores. In our experiments, we used only 2 and 4 cores of the A2 chip. However, we can scale our technique by grouping cores on a chip (e.g. groups of 4 cores), and provisioning queues to explicitly provide all-to-all communication only for cores within a group. Thus, the hardware can be configured to use a certain number of primary cores, each associated with a small number of secondary cores. An alternate design for scaling to a large number of cores limits the total number of queues implemented in hardware. In this case, the available queues can be configured by application code to serve specific pairs of sender and receiver cores. When the number of available queues is limited, we can constrain the partitioning algorithm in the compiler (Section III-B) such that generated code uses at most a specific number of queues. However, given the very fine-grained level of parallelism that we are trying to exploit, we do not expect the technique to scale to a large number of secondary cores for one primary core.

Our technique for fine-grained parallelism is independent of conventional techniques used to exploit parallelism in programs via multithreading. It can be used in conjunction with multiple hardware threads on the A2 core executing application code. However, we have not yet experimented with simultaneously using SMT[8] threads. We expect that the considerations will be similar to those applicable when normally deciding whether or not to use SMT threads (balanced use of memory and processing resources amongst the code sections executed by multiple threads). In general, our model is to first use as many hardware threads as needed for explicit user-level threads. Our technique becomes applicable when there are unused threads. Threads may be left over either because the application code did not use all available threads, or because the code is such that it can effectively utilize only a limited number of threads (due to resource constraints such as memory bandwidth).

III. Compiler Transformations

We use the compiler to automatically detect fine-grained parallelism, and to transform serial code sections into parallel versions. In our implementation, the compiler focuses on code sections that are bodies of loops that take up a significant amount of dynamic execution time. Even though we currently focus only on innermost loop bodies, the techniques described are readily extensible to generic code sections. The following sub-sections describe the sequence of compiler analyses and transformations that we have implemented within the IBM XL compiler framework.

A. Expose Fine-Grained Parallelism

We define a fiber to be a sequence of instructions without any control flow or memory carried dependences among its instructions. We partition the code into fibers, thus exposing fine-grained parallelism. The code is in the form of expression trees, and the partitioning algorithm works individually on the expression tree for each statement. This algorithm was adapted from the work in [9].

Initially, all nodes in an expression tree are unassigned to any fiber. Leaf nodes, i.e. memory loads or literal values, are treated as live-ins and they always remain unassigned. We perform a post-order traversal of the expression tree, and handle the following three cases:

- all children of the current node are unassigned: start new fiber for current node and all its non-leaf children.
- all assigned children of the current node belong to the same fiber: continue with the same fiber for the current node and all its non-leaf children.
- children of the current node are assigned to more than one fiber: start a new fiber for the current node and all its unassigned non-leaf children.

Figure 4 illustrates the tree for an example expression, and shows how the nodes are partitioned into three fibers. All the five leaf nodes remain unassigned. As internal nodes are traversed in post-order, a new fiber is started for nodes C and

D, the same fiber is continued for node B, and a new fiber is started for node A.

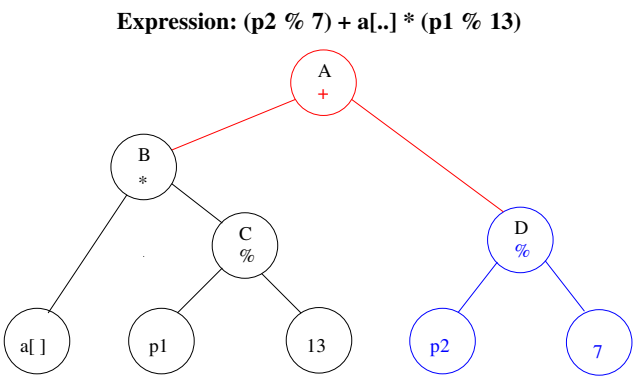

Fig. 4. An Example to Illustrate Fibers in an Expression Tree

The structure of the expression trees affects the granularity of fibers found. Before applying the partitioning algorithm, the expression trees are pre-processed to reduce the depth of the tree by splitting compound expressions into multiple statements. This makes it possible to detect even more fine-grained parallelism.

B. Determine Code Partitions for Hardware Cores

Once fibers have been identified, a graph (called the *code graph*) is built. Each node in this code graph represents a fiber. Edges between nodes represent data and control dependences between code sections that correspond to node fibers. These dependences are determined from information gathered in our compiler framework, including use-def analysis, aliasing information, and dependence vectors.

Next, the graph is transformed by merging a pair of nodes at each step, until the total number of nodes is equal to the number of hardware cores available for execution. When two nodes are merged, the resulting single node represents the fibers for both nodes, and inherits dependence edges from both nodes. Any dependence edges that may have existed between the two nodes being merged no longer exist after the merge. At the end of the transformation, each node in the graph corresponds to a hardware core, and the fibers for the node represent the code that will execute on that core. We have also implemented a different version of the merge algorithm that chooses multiple node pairs to merge at each step, instead of choosing just one pair of nodes. This version allows faster compilation, and becomes useful when there are a large number of fibers to process.

Each step of the graph transformation chooses one or more pairs of nodes to merge based on a set of heuristics. These heuristics attempt to both maximize the number of operations concurrently performed in different cores, and minimize the number of times a value has to be transferred between cores. Multiple individual heuristics are weighted and combined to compute an affinity value for each node pair. The node pair with the greatest affinity is merged, and then affinities are re-computed for the next merge step. We have experimented with many different heuristics, but the ones that worked best are:

- Assign higher affinity to node pairs with greater number of dependence edges between them.
- Assign higher affinity to node pairs with smaller compute time for the code sections they represent. The compute time is a static estimate obtained using fixed latencies for compute operations, and profile feedback data for memory access miss latencies.
- Assign higher affinity to node pairs whose code sections have greater proximity in the serial source code. This heuristic uses source code line number information to determine proximity.

We also evaluated the effect of using a throughput heuristic. This heuristic constrains partitioning to allow only unidirectional dependences between any two nodes in the final graph. As a result, there will be at least one final partition (graph node) that is independent of all other partitions, and that can execute unhindered by core-to-core communication delays. The heuristic is implemented by looking for cycles at each step in the graph transformation. If any cycles are found, then all nodes that are part of the same cycle are merged together into a single node. In our experiments, the impact of this heuristic on performance was mixed, with 3 of 18 kernels showing performance improvement, and 6 of 18 kernels showing performance degradation, and an overall slowdown of 11% on average. In general, we observed that any change in the partitioning algorithm affected performance, and very often the impact was difficult to predict, with different programs being affected in different ways.

The fibers for each node in the final graph represent code that will execute on a single core. We optimize this code by re-arranging and interleaving code instructions such that instructions producing values to be communicated to other cores execute as early as possible, and instructions that depend on values obtained from other cores execute as late as possible.

C. Outline Parallel Code

After code has been partitioned for multiple cores, the original code is transformed to outline code sections that will execute in different cores. These code sections now become separate new functions. Figure 5 uses an example to illustrate this transformation. Before outlining, there is a single function, with the instructions assigning to *T2*, *T3*, and *A* corresponding to one core, and the remaining instructions corresponding to another core. After outlining, there are three functions, the original function and two outlined functions F1 and F2. The original function has the assignment instructions deleted and replaced by calls to F1 and F2 with appropriate values passed as arguments. The functions F1 and F2 each contain one set of partitioned instructions, and F1 and F2 will execute on separate cores.

D. Insert Communication

After outlining, the compiler inserts core-to-core communication calls in the outlined code. Calls are inserted according

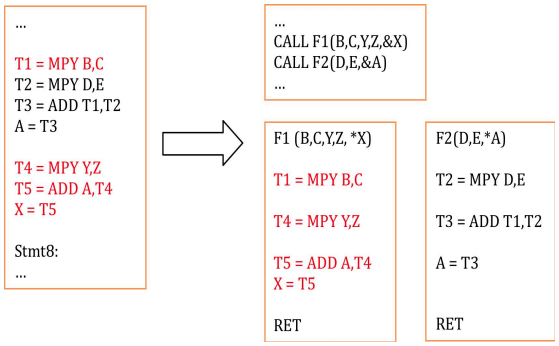

Fig. 5. Transformation to Outline Code for Multiple Cores

to dependence edges in the code graph. An Enque call is inserted after a value has been produced, in order to place it in the appropriate communication queue. A Deque call is inserted before the use of that value, in order to read it from the appropriate communication queue. Figure 6 continues the example of Figure 5, and shows the two pairs of Enque/Deque calls inserted to communicate values between functions F1 and F2. In the example, #F1 and #F2 refer to the queue identifiers corresponding to the cores on which F1 and F2 will execute.

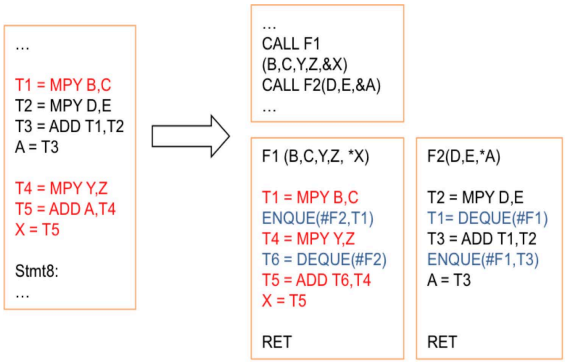

Fig. 6. Transformation to Insert Communication Calls

The Enque and Deque calls are treated as intrinsics. They get transformed into the newly defined machine instructions when the compiler backend produces binary code.

E. Replicate Conditional Structure

When the code contains branches, we need to ensure that effectively the same branch structure is reflected in all outlined functions produced due to code partitioning. Figure 7 shows an example if-then-else code snippet, where the condition variable is CND, Stmt 1 and Stmt 2 make up the then-block, and Stmt 3 and Stmt 4 make up the else-block. After partitioning, the code for computing the value of conditional CND is contained only in outlined function F1, Stmt 1 and Stmt 4 are contained in outlined function F1, and Stmt 2 and Stmt 3 are contained in outlined function F2. Thus, F1 and F2 both contain part of the then-block and part of the else-block. For correct code, we replicate the entire if-then-else structure in both F1 and F2 by copying the FJP (jump if false), JP (jump), and LAB (label) instructions, and inserting them in appropriate locations in the outlined code. Also, since the conditional is computed only in F1, we insert a pair of Enque/Deque calls to transfer the value of CND from F1 to F2. Alternatively, we could have replicated the code for computing the conditional in F2, but we have not yet experimented with the option of replicating computation in different threads.

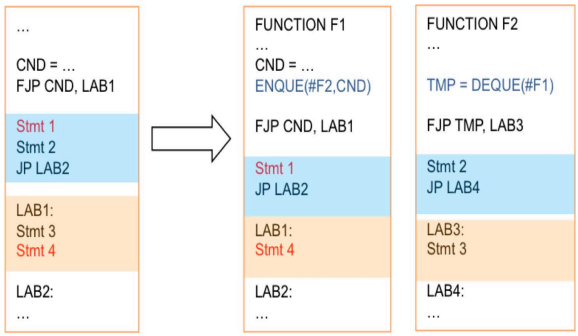

Fig. 7. Transformation to Handle Branches in Code

In general, we handle branches in the code to be partitioned by first analyzing control dependences. We compute a set of control flow predicates for each statement. A control flow predicate is a conditional variable paired with a value such that the statement can be executed only if the variable has the corresponding value at that point in the execution.

Next, code is partitioned as described in the preceding sections, and multiple outlined functions are produced as a result. During partitioning, code motion may be applied to the code within a partition to optimize it. When applying any code motion transformations, we ensure that statements that share the same control flow predicate remain grouped together. This makes it straightforward to replicate the necessary control flow structure across multiple outlined functions, by copying branch and label instructions as needed.

Finally, pairs of Enque/Deque calls are inserted to transfer the values of conditional variables that are computed in one core (outlined function), but used in one or more different cores (outlined functions).

F. Copy Live Variables

Code partitioning works only on specific code sections in the overall application, and transforms these sequential code sections into fine-grained parallel code. In our execution model, there is one primary core before entry into and after exit from such code sections. Within the code section, one or more secondary cores join the primary core for parallel execution. If any secondary core computes the value of a temporary variable that is to be later used in the primary core (after exit from the fine-grained parallel code section), this

value must be transferred to the primary core. We insert pairs of Enque/Deque calls to transfer the values of any such live variables. Figure 8 continues the example of Figure 6, and shows the Enque/Deque calls inserted to transfer the value for the live variable T2.

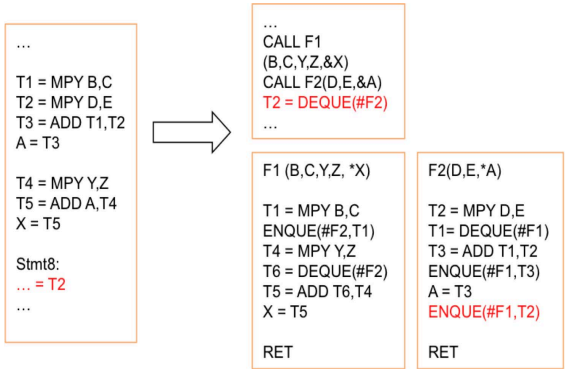

Fig. 8. Transformation to Handle Live Variables

G. Manage Runtime Threads

The compiled code links with a runtime library that includes functions to spawn threads that will execute on secondary cores (secondary threads), to terminate these secondary threads at the end of execution, and to run driver code within each secondary thread. The driver code is responsible for receiving signals from the thread executing on the primary core (the primary thread), and for launching the execution of the corresponding outlined function in response.

The compiler inserts a runtime library call at the application entry point to spawn all secondary threads. Each secondary thread loops waiting for a value to be passed to it from the primary thread (using a Deque call). When a value is received, it is interpreted to be a function pointer, and the thread branches and executes the corresponding function code. On returning from the function, the thread once again loops waiting to receive another value from the primary thread.

In the primary thread code, when an outlined function is to be executed on a secondary core, the function call is translated into an Enque call to transfer the function pointer to the secondary core. Any arguments to the function are also transferred using Enque calls, and corresponding Deque calls are inserted at the beginning of the outlined function code.

Figure 9 uses the example of Figure 8 to show the code executing in one primary thread and one secondary thread. The primary thread begins by executing the Main function, while the secondary thread begins by executing the Driver function from the runtime library. To invoke function F2, the primary thread uses Enque calls to transfer its function pointer and argument values. In the parallel code section, functions F1 and F2 execute concurrently, with F1 in the primary thread and F2 in the secondary thread. At the end of its execution, function F2 uses an Enque call to inform the primary thread of its completion. In the figure, #P refers to the queue identifier corresponding to the core on which the primary thread executes.

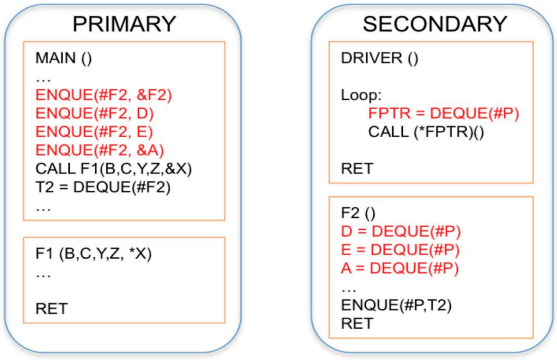

Fig. 9. Example to Show Execution of Parallel Runtime Threads

The overhead to start fine-grained parallelism is one queue transfer per thread (to pass the function pointer), and a variable (but usually very small) number of transfers to pass initial parameters. The overhead to switch back from fine-grained parallelism is a barrier (implemented using queue transfers). Also, threads are likely to be in idle mode when waiting for work to be assigned. Transitioning threads from idle to active is another overhead in starting fine-grained parallel sections. These overheads are relatively small. For the 18 loops that we experimented with, all of them execute a large number of iterations so this cost is negligible. In general, overhead costs are an important factor when selecting code sections to apply the technique to.

H. Control Flow Speculation

Speculation is a well-known technique to optimize execution time[10]. It concurrently executes parts of a serial code section, but does not commit the result of any execution until it can ensure that no serial dependencies have been violated. If any dependencies are violated, then the relevant sections of code are re-executed.

We experimented with a limited form of control flow speculation that does not need to re-execute any code. We identify if-then-else statements where the code in the then-block and else-block is mostly independent and has no side effects. This code can then be concurrently executed ahead-of-time, before the value of the conditional is known. The form of speculation we use in our transformation is very limited: it is guaranteed not to require rollback. We have deliberately used this limitation because the compiler is required to ensure that all communications will be paired at runtime, with a receiver matching a sender at each point. Allowing rollbacks in general will make it complex to handle these communications. Figure 14 in Section V shows the performance impact of using this technique.

Figure 10 illustrates our transformation for control flow speculation, using a code snippet that is a recurring pattern in the applications that we studied. In this example, Func2

and Func3 are independent functions with no side effects, while ptrVar points to a location in shared memory. After code partitioning, outlined function F1 computes the conditional value for CND and transfers it to outlined functions F2 and F3. F2 executes the then-block, and F3 executes the else-block.

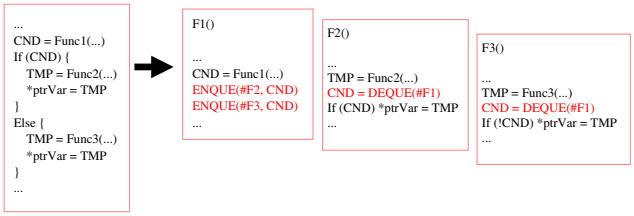

Fig. 10. Transformation for Control Flow Speculation

I. Compiler Limitations

There are three major limitations the compiler faces:

1) Correctly identifying code regions that the technique will be profitable for: We have used directives in the source code for this purpose. Alternatively, the compiler can generate multiple code versions for regions with potential, and rely on a runtime system with dynamic feedback to decide which code version to execute.
2) Memory disambiguation and dependence analysis: Accuracy of these analyses is very important since they determine the data flow, and decide which values need to be communicated between cores. Inaccuracies can increase processing time, lead to load imbalance during partitioning and cause unnecessary data exchanges. In our case, since we apply our technique within relatively small code sections, it is possible to obtain information that is accurate within the restricted scope of the code section.
3) Predicting execution time (including memory delays) for specific code segments: When partitioning code, the compiler uses several heuristics including estimated execution time. These heuristics have a large impact on performance of code generated. The compiler is unable to accurately estimate execution time, and it needs to use a profile directed feedback mechanism for this.

In addition, the compiler has to statically ensure that senders and receivers are always paired at runtime. This requirement complicates code scheduling and handling of conditionals.

IV. Code Characterization

For our evaluation, we chose the Sequoia benchmarks[5] since they are openly available, and reflect real-world applications that run on the Blue Gene/Q system. We studied the five tier 1 applications from the Sequoia benchmarks: *lammps*, *irs*, *umt2k*, *sphot*, and *amg*. These applications are already parallelized using MPI and OpenMP. We used profiling information obtained by running the parallel benchmarks on a Blue Gene machine, and focused on those functions that take greater than 2% of dynamic execution time. Within each function, we identified all innermost loops that take up a significant amount of dynamic execution time for the function. This resulted in a total of 51 such loops that were identified as hot loops. We observed that 35 of 51 hot loops are invoked from within an outer parallel loop. This reinforces the need to exploit nested and fine-grained parallelism for performance scaling on massively parallel systems.

For fine-grained parallelism, we need code sections that have multiple compute operations, with several operations that do not depend on the immediately preceding ones. We inspected the hot loops, and found that many of them are not amenable to our technique for fine-grained parallelism:

- 6 loops lack arithmetic operations: these are initialization loops that perform simple assignments to array elements.
- 25 loops are better suited to traditional loop parallelization: these loops have few arithmetic or logic operations within a loop iteration, and many of them implement a simple vector dot-product operation. Nested parallelization techniques can be applied to these loops, and the only dependences are reduction operations:
  - 8 loops perform reductions on scalar variables, which can be easily privatized for parallel execution.
  - 1 loop (in *amg*) performs reductions on array elements, which makes it more difficult to parallelize.
- 2 loops (in *umt2k*) have many conditionals in the loop body, with variables in the conditional expressions involved in read-after-write dependences. It is unprofitable to extract fine-grained parallelism within the small blocks of code between conditionals.

We are left with 18 loops that are listed in Table I, and that are used in the experimental evaluation described in Section V. Note that there are no loops from *amg* in the list. For the remaining benchmarks, these loops account for the major part of dynamic execution time: approximately 85% for *lammps*, 65% for *irs*, 50% for *umt2k*, and 55% for *sphot*.

TABLE I
List of Kernel Loops of Interest

Kernel Loop	Location in Benchmark (file, function, line number)	% Time
lammps-1	pair_eam.cpp, PairEAM::compute, line 182	30.0
lammps-2	pair_eam.cpp, PairEAM::compute, line 214	0.3
lammps-3	pair_eam.cpp, PairEAM::compute, line 247	49.5
lammps-4	neigh_half_bin.cpp, Neighbor::half_bin_newton, 172	3.6
lammps-5	neigh_half_bin.cpp, Neighbor::half_bin_newton, 199	3.6
irs-1	rmatmult3.c, rmatmult3, line 75	55.6
irs-2	MatrixSolve.c, MatrixSolveCG, line 287	5.1
irs-3	MatrixSolve.c, MatrixSolveCG, line 250	2.5
irs-4	DiffCoef.c, DiffCoef_3D, line 191	0.6
irs-5	DiffCoef.c, DiffCoef_3D, line 317	1.5
umt2k-1	snswp3d.F90, snswp3d, line 96	5.5
umt2k-2	snswp3d.F90, snswp3d, line 117	8.0
umt2k-3	snswp3d.F90, snswp3d, line 145	5.2
umt2k-4	snswp3d.F90, snswp3d, line 158	22.6
umt2k-5	snswp3d.F90, snswp3d, line 178	1.0
umt2k-6	snswp3d.F90, snswp3d, line 208	5.7
sphot-1	execute.f, execute, line 88	0.6
sphot-2	execute.f, execute, line 300	57.3

7 of the 18 loops have no conditionals within the loop body. The remaining loops either have sufficiently large independent blocks of code between conditionals, or the conditional blocks

themselves can be executed in parallel with other code sections. The one exception is the loop in umt2k-6, and as a result this loop does not show performance improvement for fine-grained parallel execution (see Section V).

Using hardware for Single Instruction Multiple Data (SIMD) execution (e.g. [11][12]) is another way to exploit fine-grained parallelism, complementary to the approach described here. The code in *lammps* and *sphot* is not suitable for SIMD. Among the 18 loops that we focused on, 2 loops show significant speedup with 4-way SIMD: irs-1 shows a speedup of 1.17, and umt2k-4 shows a speedup of 1.90. Both these loop bodies contain enough computation to be simultaneously targeted by SIMD as well as our technique for fine-grained parallelism.

V. Evaluation

We base our evaluation on the five Sequoia benchamrks descibed in Section IV. Table I lists the 18 hot loops identified from these benchmarks. Each loop is extracted into a separate kernel program, together with the necessary initialization code from the main application. All function calls are inlined in the kernel codes, and our transformation for fine-grained parallelism is applied to code in the body of each loop.

To run code generated by our compiler, we used the Mambo simulator which models the overall Blue Gene/Q system and the A2 processor pipeline with high fidelity. In the simulator, the A2 core was modified to support hardware communication queues and *enque*/*deque* instructions (as described in Section II). Processing an *enque* or *deque* instruction takes one cycle in the processor pipeline. Additionally, there is a queue transfer latency, which is the minimum number of cycles that must elapse before a value written into a queue by one core becomes accessible in another core. Figure 11 illustrates the effect of this transfer latency. In the figure, time proceeds from top to bottom, and core 1 issues an *enque* instruction at time T_A. If core 2 issues a corresponding *deque* instruction earlier at time T_B, then processing in core 2 is stalled until time (T_A+transfer latency). On the other hand, if core 3 issues a corresponding *deque* instruction at time T_C, which is later than time (T_A+transfer latency), then processing in core 3 resumes immediately.

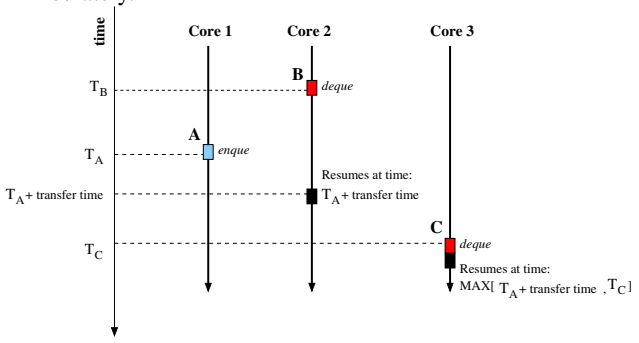

Fig. 11. Effect of Transfer Latency on Core-to-Core Communication

There are separate queues for floating point values and for general-purpose register values. The queues have a fixed length, and *enque*/*deque* instructions block when there are no empty slots to write into, or no available values to read from a queue. The simulator supports point-to-point communication queues for four A2 cores, and it parameterizes the queue length as well as the queue transfer latency. The queue length is set to 20 slots, and the transfer latency is set to 5 cycles by default.

Figure 12 shows, for each of our kernel benchmarks, the speedup over sequential execution time that is observed when running fine-grained parallel code with 2 cores or 4 cores. Note that the base sequential version in our experiments corresponds to nested innermost loops in applications that have already been parallelized at the outer levels using MPI and OpenMP. For 2 cores, the speedups range from 1.03 to 1.76, with an average speedup of 1.32. For 4 cores, the speedups range from 0.90 to 2.98, with an average speedup of 2.05. We did not observe any speedup for umt2k-6. Table II uses the data in Table I and Figure 12 to show expected speedups for whole applications.

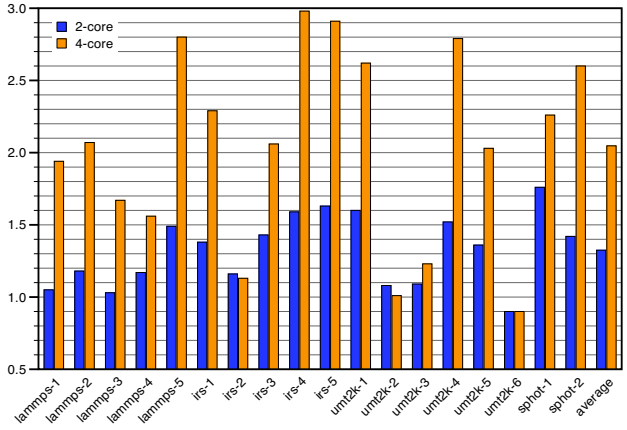

Fig. 12. Speedup of Fine-Grained Parallel Code Over Sequential Code

TABLE II
Speedups for Whole Applications

Application	2-core	4-core
lammps	1.05	1.70
irs	1.24	1.79
umt2k	1.16	1.51
sphot	1.25	1.92
average	1.18	1.73

For each kernel loop, Table III gives the following numbers for the 4-core case:

- Initial Fibers: the number of fibers initially found in the code of the loop body.
- Data Deps: the number of data dependences between the initial fibers found.
- Load Balance: this is determined by examining the number of compute operations in each of the 4 fine-grained threads. The number shown is the ratio of the largest number of compute operations in a thread to the smallest number of compute operations in a thread. A ratio of 1 implies the code is perfectly balanced. We observe reasonable numbers for all kernels except umt2k-2 and

umt2k-3. The loop body for both these kernels contains only reduction statements within conditionals.

- Com Ops: the number of enqueue/dequeue operations inserted in the transformed code, to transfer either data or control dependence values between threads. This number gives the number of times data transfers occur in the code.
- Queues: the number of queues for distinct sender-receiver pairs of cores that are actually used to transfer values during execution. Note that core A sending to core B and core B sending to core A count as 2 separate queues. The number of queues used is generally low, with a maximum of 8 queues being used.
- Speedup: obtained over sequential execution, when executing transformed code using 4 cores.

TABLE III
DETAILED EXPERIMENTAL DATA (FOR 4 CORES)

Kernel	Initial Fibers	Data Deps	Load Bal	Com Ops	Num Ques	Spdup
lammps-1	63	37	1.49	9	3	1.94
lammps-2	60	6	1.89	6	3	2.07
lammps-3	123	96	1.49	23	6	1.67
lammps-4	105	67	1.68	34	6	1.56
lammps-5	87	14	1.45	18	6	2.80
irs-1	268	54	1.69	3	3	2.29
irs-2	47	6	2.54	8	6	1.13
irs-3	9	3	1.88	2	2	2.06
irs-4	110	108	1.65	16	3	2.98
irs-5	390	698	1.84	60	3	2.91
umt2k-1	11	6	1.91	2	2	2.62
umt2k-2	33	2	87.50	3	2	1.01
umt2k-3	31	4	35.00	5	3	1.23
umt2k-4	35	62	1.67	10	7	2.79
umt2k-5	9	28	1.3	6	6	2.03
umt2k-6	38	1	1.57	6	6	0.90
sphot-1	5	2	2.36	2	2	2.26
sphot-2	478	329	1.71	36	8	2.60

We experimented with changing the queue transfer latency from 5 cycles to 20 cycles and 50 cycles. Figure 13 shows the resulting degradation in performance for the 4-core parallel version. Overall, for 20 cycles, there is about 20% performance degradation, and the average speedup decreases from 2.05 to 1.85. Also, there are four kernels that show no speedup when the transfer latency is 20 cycles: umt2k-6, umt2k-2, irs-2, and lammps-4. For 50 cycles, the average speedup is 1.36, and 6 kernels show no performance speedup. If the latency is further increased to 100 cycles, there is no speedup on average, and only 2 of 18 kernels show a speedup (irs-1 and irs-4). The technique is inherently sensitive to communication latencies, and we do not expect gains at higher latencies.

Figure 14 shows the effect of optimizing code by applying control flow speculation, as described in Section III-H. This optimization improves the performance of eight kernels, resulting in an overall increase in performance of about 28%, with the average speedup improving from 2.05 to 2.33.

VI. RELATED WORK

Balakrishnan et. al.[13] evaluate the merits of using an architecture with a few strong cores and many simpler cores. They show that while the strong cores improve the performance of sequential code regions, the overall performance

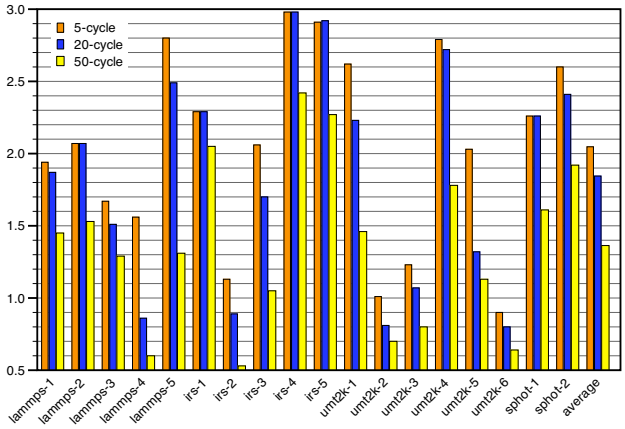

Fig. 13. Effect of Queue Transfer Latency on Speedup on 4 Cores

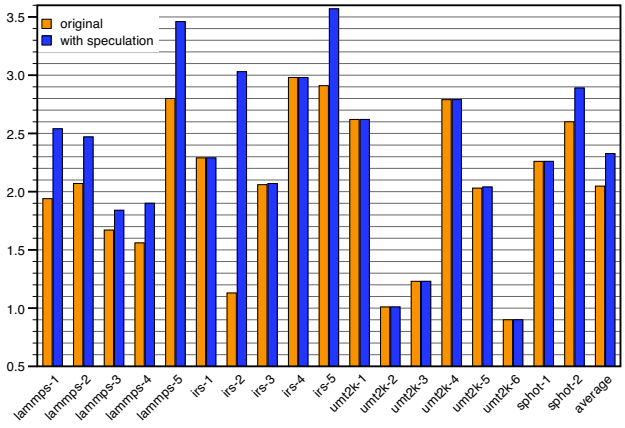

Fig. 14. Effect of Control Flow Speculation

can become unpredictable, and scalability can be limited by the weakest core in the system. Given the advantages of scalability, reliability and power efficiency of a design using simpler homogeneous cores, the work in [4] advocates using a core fusion approach instead. In core fusion, multiple simpler cores can either work independently when executing parallel code sections, or they can switch to a resource-sharing mode where several cores together approximate the behavior of a single powerful superscalar core. The results in [4] indicate that on their architecture, sequential codes perform as well as, while parallel codes perform better than, a 4-way superscalar core using comparable resources. However, since the solution is hardware-based only, it cannot consider the whole application code when looking for fine-grained parallelism, or use apriori knowledge of dependences in the code. Kim et. al.[14] describe another core fusion solution based on the TRIPS architecture[15], which is similar to our approach in that it relies on the compiler to detect concurrency and make the data flow explicit. However, our approach works on conventional

instruction set architectures, while the work in [14] relies on the custom ISA supported only by TRIPS.

Madriles et. al.[16] describe a combined hardware-software solution for improving the performance of sequential code regions using fine-grained multi-threading. This combined solution results in approximately 10% performance improvement over an approach similar to the core fusion described in [4]. Their work uses a compiler to partition sequential code into threads, and special "Program Order Pointer" bits to encode the sequential order in generated code. The code partitioning is aggressively speculative, ignoring memory dependences when creating threads. During execution, special hardware units use information about the sequential order to check for memory violations and roll back when necessary. This approach relies on speculative execution and hardware conflict detection/rollback, which can negatively impact the energy efficiency. Also, the compiler partitioning algorithm used is conservative with respect to control flow dependences.

DSWP[17] and OUTRIDER[18] take an approach similar to ours, using software to extract fine-grained threads and special hardware support for inter-core communication. OUTRIDER is different in that it focuses on using fine-grained threads to hide memory latency, while we consider both memory latencies and compute operations when partitioning code. The DSWP approach is closest to ours, but the process used for code partitioning and the evaluation contexts are very different. We use a Blue Gene/Q system and high-performance computing benchmarks, and evaluate for both 2-threads and 4-threads. Further, our partitioning does not restrict fine-grained threads to have no cyclic dependences. We found that this restriction results in a slowdown of 11% on average in our system (see Section III-B). Also, we obtain qualitatively different results in our experiments, observing high sensitivity to the communication latency between threads. The follow-up work done to optimize DSWP[19][20] is complementary to our work, and can be applied to our technique as well.

VII. Conclusion

We have described a compiler technique for automatically transforming sequential code into fine-grained parallel code. We have also presented hardware features needed to realize the performance potential when using this technique. We implemented the compiler transformations in the IBM XL compiler framework and the hardware features in the Mambo BlueGene/Q simulator. Since fine-grained parallelism becomes useful when coarse-grained parallelism has exhausted its limits, we characterized the five Sequoia tier 1 applications to discover code regions where our technique applies. We experimented with these code regions, and evaluated the performance obtained. On 2 cores, we observed an average speedup of 1.32 (ranging from 1.03 to 1.76), and on 4 cores, we observed an average speedup of 2.05 (ranging from 0.90 to 2.98). Thus, our technique shows potential for improved performance scaling, and for exploiting unused cores in the system.

Acknowledgments

We thank Robert Walkup for providing us with application profiling data, and Tong Chen for the data on SIMD performance. This work was supported in part by the Lawrence Livermore National Laboratory under contract B554331.

References

[1] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[2] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 2.2," September 4th 2009, available at: http://www.mpi-forum.org.

[3] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[4] E. Ipek, M. Kirman, N. Kirman, and J. Martinez, "Core fusion: Accommodating software diversity in chip multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[5] Lawrence Livermore National Laboratory, "ASC Sequoia Benchmark Codes." [Online]. Available: http://asc.llnl.gov/sequoia/benchmarks

[6] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012.

[7] D. S. Gallo, J. R. Brunheroto, and K. D. Ryu, "Fast full-system execution-driven performance simulator for Blue Gene/Q," in *Euro-Par*, 2013.

[8] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *International Symposium on Computer Architecture(ISCA)*, 1998.

[9] H.-S. Kim and J. E. Smith, "An instruction set and microarchitecture for instruction level distributed processing," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[10] V. Krishnan and J. Torrellas, "Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor," in *Proc. of 1998 Int. Conf. on Supercomputing*, 1998, pp. 85–92.

[11] K. Diefendorff, P. Dubey, R. Hochsprung, and H. Scales, "AltiVec Extension to PowerPC Accelerates Media Processing," *IEEE Micro*, Mar. 2000.

[12] Intel Corporation, "Intel Advanced Vector Extensions Programming Reference," June 2011, reference 319433-011.

[13] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *International Symposium on Computer Architecture (ISCA)*, 2005.

[14] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. Keckler, "Composable lightweight processors," in *International Symposium of Microarchitecture*, 2007.

[15] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *International Symposium on Computer Architecture(ISCA)*. ACM Press, 2003.

[16] C. Madriles, P. Lopez, J. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez, "Boosting single-thread performance in multi-core systems through fine-grain multi-threading," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[17] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *International Symposium on Microarchitecture(MICRO)*, 2005.

[18] N. C. Crago and S. J. Patel, "Outrider: efficient memory latency tolerance with decoupled strands," in *International Symposium on Computer Architecture(ISCA)*, 2011.

[19] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. August, "Parallel-stage decoupled software pipelining," in *International Symposium on Code Generation and Optimization*, 2008.

[20] J. Huang, A. Raman, T. Jablin, Y. Zhang, T.-H. Hung, and D. August, "Decoupled software pipelining creates parallelization opportunities," in *International Symposium on Code Generation and Optimization*, 2010.

[21] F. Tseng and Y. N. Patt, "Achieving out-of-order performance with almost in-order complexity," in *International Symposium on Computer Architecture (ISCA)*, 2008.

[22] S. Kumar, C. J. Hughes, and A. D. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *International Symposium on Computer Architecture(ISCA)*, 2007.