Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads

Teo Milanez, Caroline Collange, Fernando Magno Quintão Pereira, Wagner Meira, Renato A. Ferreira

Thread scheduling and memory coalescing for dynamic vectorization of SPMD workloads

Teo Milanez[a], Caroline Collange[b], Fernando Magno Quintão Pereira[a], Wagner Meira Jr.[a], Renato Ferreira[a]

[a] *Universidade Federal de Minas Gerais*
[b] *INRIA*

Abstract

Simultaneous Multi-Threading (SMT) is a hardware model in which different threads share the same processing unit. This model is a compromise between high parallelism and low hardware cost. Minimal Multi-Threading (MMT) is one architecture recently proposed that shares instruction decoding and execution between threads running the same program in an SMT processor, thereby generalizing the approach followed by Graphics Processing Units to general-purpose processors. In this paper we propose new ways to expose redundancies in the MMT execution model. First, we propose and evaluate a new thread reconvergence heuristic that handles function calls better than previous approaches. Our heuristic only inspects the program counter and the stack frame to reconverge threads; hence, it is amenable to efficient and inexpensive hardware implementation. Second, we demonstrate that this heuristic is able to reveal the existence of substantial regularity in inter-thread memory access patterns. We validate our results on data-parallel applications from the PARSEC and SPLASH suites. Our new reconvergence heuristic increases the throughput of our MMT model by 7%, when compared to a previous, and substantially more complex approach, due to Long *et al.* Moreover, it gives us an effective way to increase regularity in memory accesses. We have observed that over 70% of simultaneous memory accesses are either the same for all the threads, or are affine expressions of the thread identifier. This observation motivates the design of newly proposed hardware that benefits from regularity in inter-thread memory accesses.

Keywords:
Static analysis, divergence analysis, SIMD, MMT, high performance

1. Introduction

Graphics Processing Units (GPUs) have introduced an execution model based on lockstep execution of multiple threads running the same program. This model joins the efficiency of SIMD execution, with the programmability of multi-thread processors. It has inspired several recent works that generalize the GPU approach to general-purpose CPUs. Thread Fusion [1], Minimal Multi-Threading (MMT) [2] and Multi-thread Instruction Sharing (MIS) [3] are techniques that share the decoding and execution of identical instructions between threads. In this paper, we will refer to these architectures as MMT for conciseness, although our discussion applies to any architecture exploiting instruction redundancy across threads. An MMT-based architecture organizes threads in groups that share the instruction fetch logic, and might share execution units. Each thread keeps its own program counter (PC). At fetch time, the hardware chooses heuristically the next PC to serve. If the chosen PC is the same across several threads, then these threads receive the same instruction to execute. If the input values are the same for all threads executing the instruction, then the computations are combined too, so the instruction is issued once on behalf of all participating threads.

Minimal Multi-Threading, being a recent notion, still offers room for improvements. In particular, Thread Fusion needs explicit synchronization points inserted by the compiler [1]. MIS relies on incidental synchronization between threads after barriers, but does try to maintain synchronization. Long *et al.*'s original formulation of MMT uses an intricate reconvergence heuristics, which, in the words of the authors themselves, has impact on the hardware's performance [2]. This heuristic is expensive because it looks up the program counter's history at every execution cycle. Techniques used in current GPUs require specialized instruction sets with explicit reconvergence annotations [4]. Thus, they cannot be used directly on general-purpose binary programs. Techniques based on program order such as Thread Frontiers, recently proposed for GPUs [5], cannot cope well with function calls, as we show in this paper. In addition to these shortcomings, MMT, in its original conception, does not explore any form of redundancies in memory access patterns. We believe the reason for this limitation is simply the fact that researchers have not yet demonstrated that such redundancies are common in the Single Program, Multiple Data (SPMD) scenario. Nevertheless, this type of redundancy has been already acknowledged, in the GPU world, as a promising way to save hardware space and to reduce energy consumption [4].

The objective of this paper is to advance the research on MMT-based architectures, a task that we accomplish in two ways. First, we propose a new heuristic to keep threads synchronized and compare its performance against some algorithms that are already well established in the industry and in the academia. Second, we provide an analysis of memory access patterns of typical applications to motivate new designs of data fetching units. We draw the conclusions that we present in this paper from the simulation of twelve benchmarks: four data-parallel applications found in the PARSEC suite [6], seven in the SPLASH-2 suite[1], and the Tachyon Raytracer [7]. We focus on data-parallel applications, in which threads execute the same program, following Darema's SPMD model [8]. Hence, we have more opportunities to share instructions among threads. These experiments are meaningful because the benchmark programs offer remarkably complex and divergent control flow structures that have not been specifically optimized for GPU execution.

Our first contribution is a thread reconvergence heuristic that improves data-level parallelism by providing more opportunities for lockstep execution. Keeping threads as much synchronized as possible is important: if two separate threads read different program counters, then they will compete for the shared pipeline front-end, causing pipeline stalls and/or increased energy consumption. Our new heuristic gives priority to the thread with the minimum relative stack-pointer. In case of ties, it then uses the minimum PC criterion, a heuristic adapted from a policy originally proposed by Quinn *et al.* [9]. We call our approach min-SP/PC. We have compared min-SP/PC against Quinn's policy, Long's original heuristic, and 2-stack, a recent reconvergence method due to Lee *et al.* [10]. The min-SP/PC heuristic is twice more effective than min-PC and 2-stack. It is 6.5% more effective than Long's heuristic; furthermore, it admits a much cheaper implementation, be it in hardware or in software.

As a second contribution of this work, we analyze the *memory access patterns* of threads synchronized with the Min-SP/PC heuristic. Such patterns describe the relative arrangement of addresses in the load and store instructions used by each thread. We identify three different access patterns: *uniform*, *strided* and *scattered*, which we shall define in Section 5.1. We have observed substantial regularity in inter-thread access patterns. These forms of regularity have not been previously noticed because Long *et al.* [2]

[1] `http://www.capsl.udel.edu/splash/`

considered multi-process workloads, whereas we are analyzing multi-thread programs sharing a single address space. This fact motivates the adoption, in the MMT world, of recent *memory coalescing* hardware mechanisms that have been proposed for GPUs [4]. For instance, if all the threads read data from the same location, or from consecutive locations, then the hardware can bring all this data to registers with only one cache access. Patterns in which threads access uniform or regularly spaced addresses happen in over 70% of the memory accesses that we have observed.

This paper extends an earlier work [11], originally published in the 24th International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2012). Out main additions lay in the experimental section. Whereas originally we only compare our new heuristic with Quinn's algorithm, now we match it also against Long's and Lee's techniques. Furthermore, we have expanded our benchmark suite, adding to the four PARSEC applications (Blacksholes, Swaptions, Bodytrack and Fluidanimate) eight well-known programs of similar size and complexity.

2. Background

Resource multiplexing and sharing are not new ideas, but the success of GPUs has drawn renewed attention to inter-thread resource sharing. In the mid nineties Tullsen *et al.* introduced the notion of *Simultaneous Multi-Threading* (SMT) [12, 13]. In the SMT execution model, several threads share the same superscalar pipeline, including the front-end fetching and decoding instructions. In this way, the hardware is better equipped to avoid control and data hazards; hence, keeping the many stages of its pipeline always in use. The SMT hardware is present in a number of architectures, such as the Hyper-threading technology, which was introduced in Intel's Pentium 4. In 2008, Gonzalez *et al.* brought in the concept of *Thread Fusion* as a way to decrease the energy consumption of SMT machines [1]. Thread fusion consists in sharing the processor front-end to distribute the same instruction to different threads, whenever they have the same program counter. In order to reconverge threads, Gonzalez *et al.* would require the compiler to insert barriers at control independent program points, in the same way as GPUs do [4]. Long *et al.* [2] and Dechene *et al.* [3] extend Gonzalez's work by introducing the idea of also sharing the execution of instructions with identical input data. Unlike previous works, these architectures can run existing SPMD applications compiled with conventional compilers and intruction sets.

Whereas SMT implies resource multiplexing, MMT strives for resource sharing. That is, in the former case, only one thread can use a given resource at a given time. In the latter, several threads cooperatively use a resource to perform the same action, promising higher energy reductions than what could be achieved with independent thread execution. An MMT-based architecture organizes threads into groups that share the same instruction fetch and decode unit, and might share execution units. Each thread keeps its own program counter (PC). Each cycle, the hardware will select the next instruction to fetch among the program counters of active threads. The next instruction to be processed will be fetched at this PC. If the chosen PC is the same across several threads, then all of these threads receive an instruction to execute. If this instruction has the same input values, then the computations might be combined as well, so the instruction is issued once on behalf of all participating threads.

2.1. Maximum Instruction Sharing

The MMT hardware shares instructions between threads whenever these threads read from the same program counter. Thus, threads must be kept as synchronized as possible. However, this is a very difficult task, because to solve it optimally we would have to deal with an NP-complete problem: the *Shortest Common Supersequence* [14]. This problem can be stated as follows: *"given two or more sequences of symbols, find the smallest supersequence that includes all those sequences. The solution must keep the relative order of the symbols in each sequence"*. We have rephrased this problem in our context, as the maximum instruction sharing problem, which we define as follows:

Definition 2.1. Maximal Instruction Sharing

Instance: *an alphabet of opcodes* $\Sigma = \{\sigma_0, \ldots, \sigma_m\}$*, plus a set* $\{t_0, t_1, \ldots, t_n\}$ *of finite strings ranging on* Σ*, e.g.,* $t_i[j] = \sigma_k$*.*

Problem: *find the shortest string* T *ranging on* Σ*, with the following properties:*

1. ***totality:*** *for any* i, j*, there exists* x *such that* $t_i[j] = T[x]$*.*
2. ***ordering:*** *for any* i, u, v*, there exists* x, y*, such that if* $T[x] = t_i[u]$*,* $T[y] = t_i[v]$*, and* $v > u$*, then* $y > x$*.*

Each t_i represents the sequence of instructions executed by a thread; hence we call it an *execution trace*. Figure 1 illustrates an instance of instruction sharing. In this example we want to match four sequences, e.g.:

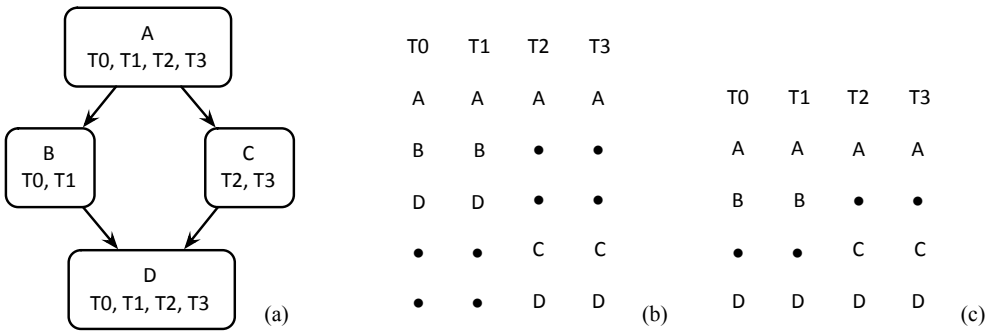

Figure 1: (a) A program with four basic blocks. (b) a sub-optimal alignment of blocks. (c) an optimal alignment of blocks.

two instances of $[A, B, D]$, and two instances of $[A, C, D]$. These sequences represent the paths that threads took when executing the program in Figure 1. We have four threads executing this program: T_1, T_2, T_3 and T_4. Block A is a *divergent point*: it contains a conditional branch, which give threads the opportunity to take different paths along the program code. Figure 1 (b) shows a scenario in which threads, after diverging, no longer resume lock-step execution. This is a sub-optimal alignment. Figure 1 (c) shows a situation in which the diverging threads synchronize at block D. This is an optimal solution to maximal instruction sharing.

We consider two versions of maximal sharing: off-line and on-line. The off-line version of this problem is equivalent to the *shortest common supersequence*. However, we are more interested in the on-line version of maximal sharing. In this case, each trace t is seen as a *queue*, with tail at $t[0]$. We can only perform one of two operations on each t_i: "inspect head t_i" and "pop t_i". The first operation lets us see the opcode at the head of t_i. The second removes the opcode from the head of t_i, and places it at the tail of T. Notice that we are not allowed to store opcodes before inserting them into T. Given that an optimal solution to the on-line version of the problem would solve its offline counterpart, we know that on-line instruction sharing is NP-complete. Therefore, this problem must be tackled by heuristics.

2.2. Thread Synchronization Heuristics

The main goal of our work is to design new thread reconvergence heuristics, and to compare these new approaches against older methods. Many different reconvergence algorithms have been described in the literature, and in this section we present a few of them. Some of the methods that we discuss here have been designed for the early SIMD machines. Others are more recent, and have been developed for modern Graphics Processing Units. In this paper, we focus on heuristics that **do not** require compiler support.

In 1988, Quinn [9] briefly mentions the idea of executing instructions in static program order, by giving priority to the "textually earliest program label". This heuristic was originally proposed in the context of a compiler of SIMD programs to MIMD machines. Its idea is to prioritize statements according to the order in which they appear in the source code, assuming reconvergence points are encountered below divergence points. In a hardware implementation, it may be implemented by comparing the PC of each thread against a global PC to find active threads. Reconvergence happens when the PCs of different threads coincide. Recently, Lee *et al.* [10] have proposed a new reconvergence algorithm that handles loops with `continue` commands better than Quinn's original method. This method has been originally called *2-stack*. Lee *et al.* also give priority to threads with the smallest PC. However, it separates threads into two stacks, which we shall call *current* and *future*. At each cycle, the threads tied with the smallest PC, in the current list, execute. Threads that take backward branches are placed in the future list. Once the current list becomes empty, these structures are swapped. The key idea behind Lee's technique is to avoid that threads that branch back early during the execution of a loop be treated as if they were slower than threads that are still iterating through that loop.

The last heuristic that we have experimented with was proposed by Long *et al.* for the MMT architecture [2]. This architecture keeps a fetch history for each thread. The thread scheduler uses the history to detect reconvergence dynamically. It is built around the idea that, if the current PC of a thread t_0 is found in the fetch history of another thread t_1, then t_0 is probably behind t_1. Thus, t_0 is given priority, based on the assumption that it needs to advance to catch up with t_1. When no match is found, threads run concurrently as in a conventional SMT processor.

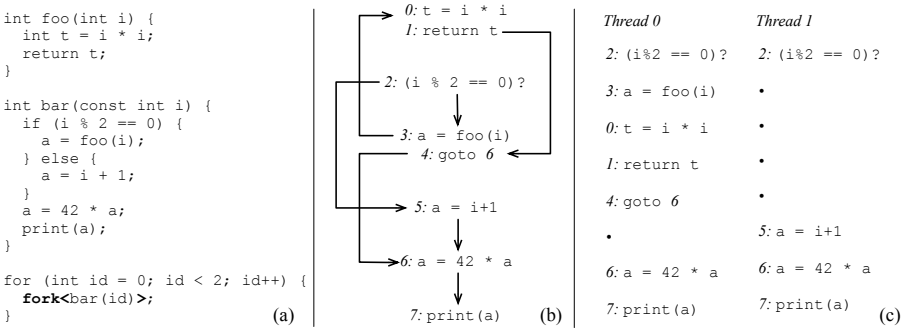

Figure 2: A example where the min-PC heuristics is able to reconverge optimally divergent threads. In this example, Min-SP/PC and Min-PC behave in the same way.

3. The Min-SP/PC Heuristic

The main contribution of this paper is a new reconvergence heuristic, which does not require compiler support, and is not only less expensive, but also more effective, than related algorithms. We call this heuristic Min-SP/PC, because it gives priorities to threads with the minimum relative stack pointer, using the minimum PC as the criterion to break ties. Our goal is to handle in a better way applications written in programming languages that feature function calls. Indeed, policies like Min-PC that embed priorities in the static program order yield a total order inside functions, but do not provide a sensible order across functions. They are thus not suitable for structured programs with function calls.

Computer architectures usually provide a *stack pointer* (SP) register to track the data manipulated by the current active function. We propose to use this value, combined with the program counter, to reconverge threads. Priority is always given to the thread with the lowest string "SP:PC" in lexicographic order. Assuming a conventional stack that grows toward lower addresses, this policy gives priority to the most inward call nesting level. In this way, if a function f calls a function g, threads that must execute code in g receive priority over the threads still executing f. As the policy is based on the dynamic value of the stack pointer, it does not depend upon the static code location in memory, and can handle mutually recursive functions. In this section we explain why our heuristic, albeit simple, is an improvement over previous work.

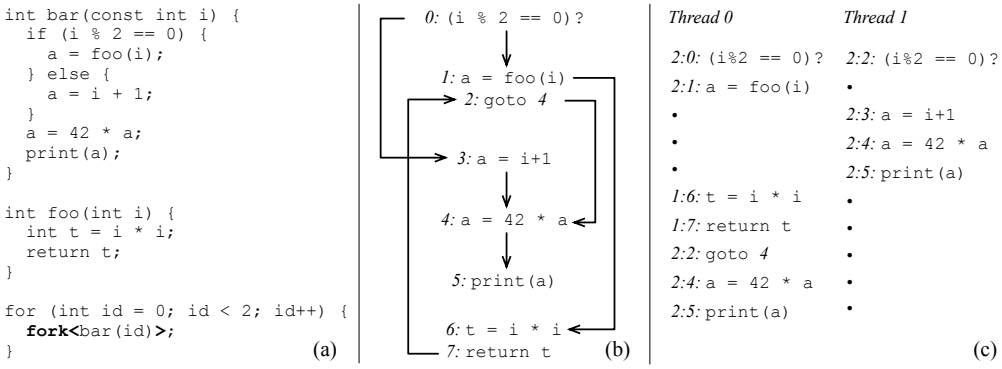

Figure 3: Example where the min-PC criterion fails to reconverge divergence threads, but the min-SP/PC does it optimally. In part (c) we prefix each instruction with its stack pointer plus program counter.

Min-SP/PC vs Min-PC: We start by comparing our algorithm to the even simpler Min-PC method, adapted from Quinn *et al* [9]. Figure 2 illustrates the min-PC heuristic. In this example and in the next one, we assume an MMT hardware that supports two threads executing simultaneously the same instruction. The program in Figure 2(a) is written in a C-like language with a special keyword, **fork**, that passes a function to a freshly created thread. Figure 2(b) provides a static view of the code that will be executed by each thread. In this rather artificial example, thread zero will follow the "then" path after the branch, whereas thread one will follow the "else" path. Thus, the execution diverges after the test at instruction 3. In face of a divergence, the min-PC heuristic keeps feeding the thread that is reading instructions at the lowest program counter. Intuitively, we expect a path from a lower PC p_l to a higher PC p_h. Therefore, the thread t_l that is at p_l is supposed to catch up with a thread t_h that is at p_h, if t_l is given priority over t_h. In this example, shared instruction fetching resumes at instruction six, and both, Min-PC, and Min-SP/PC behave in the same way.

Unfortunately, the min-PC heuristic might take too long to reconverge threads in code that contains function calls. Figure 3 illustrates this phenomenon. In this new example, function `bar` has been laid out before function `foo` in the program text. The call to `foo` happens inside a conditional when only one thread is active. Because `foo` is located after `bar`, thread one will finish `bar` before thread zero has a chance to execute `foo`. In this

case, the resulting execution trace is two instructions longer than that sequence seen in Figure 2(c). The min-PC heuristic produces a trace with ten instructions. On the other hand, the min-SP/PC heuristic would produce an optimal trace with eight instructions, because it would be able to share program counters 4 and 5.

Min-SP/PC vs 2-Stack: The reconvergence method proposed by Lee *et al.* suffers from the same problems that affect Min-PC concerning function calls. Thus, 2-stack would not be able to find the optimal instruction sharing in Figure 3. Furthermore, we have observed that while this heuristic handles loops with `continue` statements better than Min-SP/PC, it does not deal well with loops featuring `break` commands. Figure 4 illustrates this issue. If a thread finishes an iteration early, due to `continue`, then it is placed in the future list. Thus, this thread will wait for the others at the loop head. When the other threads finish the iteration, the heuristic synchronizes them at the beginning of the loop. We show this scenario in Figure 4 (c). Comparing this trace with that one produced by Min-SP/PC (or Min-PC), in Figure 4 (d), we see the advantage of Lee's approach. However, this heuristic fares poorly if threads execute different number of iterations. The threads that finished the loop earlier will continue executing, while the other threads will be waiting in the future list. This last issue can be seen in Figure 4 (e) and (f). In this case, Min-SP/PC synchronizes the threads at the end of the loop, whereas 2-stack cannot do it.

Min-SP/PC vs Long's Heuristic: The main drawback of Long's heuristic is its high implementation cost. The memory required to store the FHB is $O(n \times s)$, where s is the table's size, and n is the number of threads. The heuristic also imposes an $O(\ln n)$ delay per cycle, because each thread must check if its PC is in the table of other threads. Finally, this heuristic has a cost in terms of energy consumption, because we have $O(n \times n)$ queries being performed on the FHB. Furthermore, from an algorithm point of view, Long's heuristic faces difficulties to deal with some specific scenarios. In particular, the size of the table cannot be either too large or too small. A small table does not offer enough space to contain intersections in the paths of divergent threads. However, a large table might lead to false positives, because a thread that is ahead may find its PC in the table of a thread that is behind, due to loops. For instance, in Figure 5 (a) we might give priority to thread T_1 over T_0, because basic block C, which is visited by T_0 a second time, is in the history table of T_1.

Long's heuristic is better than 2-stacks or Min-PC at dealing with func-

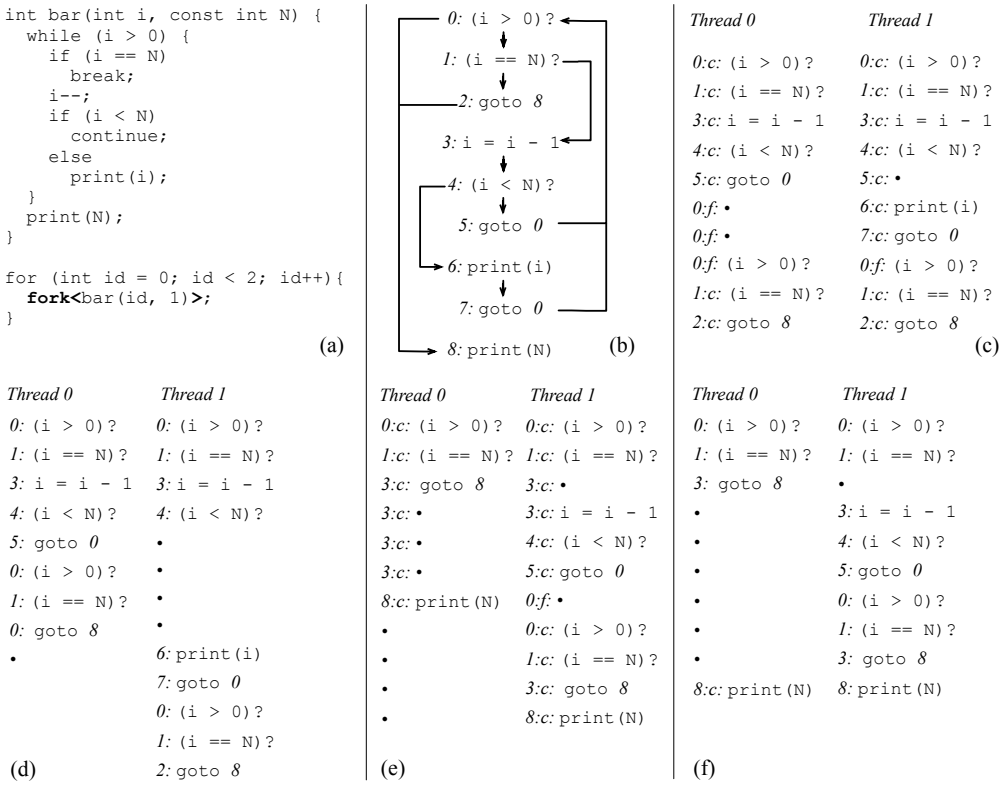

Figure 4: (a) Program in high-level language. (b) Control flow graph. (c) Thread 0 takes `continue` at label 5: 2-stack reconvergence. (d) Thread 0 takes `continue` at label 5: Min-SP/PC reconvergence. (e) Thread 0 leaves the loop at label 3: 2-stack reconvergence. (f) Thread 0 leaves the loop at label 3: Min-SP/PC reconvergence.

tion calls. Figure 5 (b) shows a program that benefits from this positive behavior. The heuristic might conclude that threads T_0 and T_1 are traversing the same code, once they enter function `foo`. On the other hand, Min-SP/PC is still better at dealing with functions than Long's approach. For instance, Min-SP/PC will synchronize threads T_0 and T_1 at block C of Figure 5. On the other hand, Long's heuristic does not provide such a guarantee.

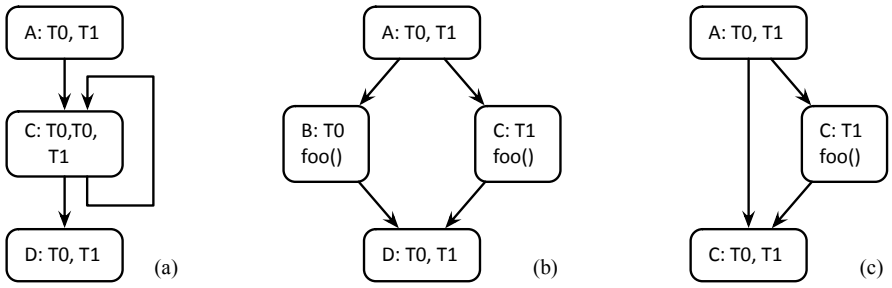

Figure 5: (a) Example that might cause false positives in Long's heuristic. (b) Scenario in which Long's heuristic is able to synchronize flow at function calls. (c) Scenario in which a function call might pose a problem to Long's algorithm.

4. Methodology

In this section we describe the methodology that we have used to compare the different heuristics. We will describe the benchmarks that we have gathered, and the infra-structure that we have used to perform this comparison.
Benchmarks: We perform the evaluation on SPMD benchmarks that run multiple threads inside a single process. We use the following benchmarks.

- Four benchmarks from **Parsec** [6] package:
  - **blackscholes**: option pricing with Black-Scholes Partial Differential Equation (PDE).
  - **swaptions**: application that uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions.
  - **bodytrack**: computer vision application that tracks a human body with multiple cameras through an image sequence.
  - **fluidanimate**: fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method.
- Seven benchmarks from **Splash-2** package:
  - **barnes**: simulation of the interaction of a system of particles with the Barnes-Hut method.

Bench	Insts	Seq	Crit	Bench	Insts	Seq	Crit
blackscholes	787	533	0	bodytrack	21,067	1,026	65,537
fluidanimate	5,274	5,691	5,856	tachyon	5,796	542	1,346
barnes	4,164	1,414	613,942	fft	2,004	697	2,102
fmm	7,289	2,668	6,749,323	ocean_ncp	11,726	379	36,040
radix	1,627	734	2,299	volrend	5,014	182	20,663
water_nsq	6,413	254	82,875	swaptions	2,248	1,123	0

Figure 6: Characteristics of the benchmarks: number of static X86 instructions, size of the dynamic sequence (in millions of instructions) and number of instructions in critical sections.

  - **fmm**: simulation of the interaction of a system of particles with a parallel adaptive Fast Multipole Method to simulate the interaction of a system of paticles.
  - **fft**: signal processing application that uses Fast Fourier Transform.
  - **radix**: radix sort algorithm.
  - **volrend**: raytracer algorithm.
  - **ocean_ncp**: simulation of large-scale ocean movements based on eddy and boundary currents.
  - **water_nsquared**: simulation of water molecular dynamics.

- **Tachyon** raytracer [7]

Figure 6 shows some characteristics of these benchmarks. We have compiled them to a 64-bit x86 architecture; hence, by "number of instructions" we mean "number of x86 instructions". We obtain the dynamic traces, i.e., the number of executed instructions, by feeding each application with its default input.

Simulation: We have instrumented the binary programs using the *Pin* framework[2]. All the traces that we produce in this paper are obtained from the execution of these instrumented programs.

The Available Thread-Level Parallelism (TLP): The more opportunities for parallel execution an application presents, the more effectively it

[2] `http://www.pintool.org/`

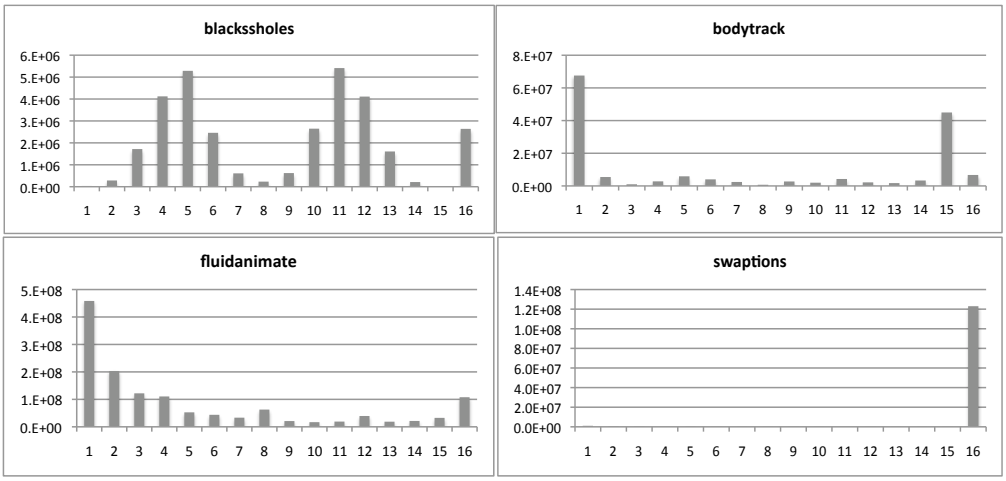

Figure 7: Histogram showing the number of cycles in which n threads were active, where $1 \leq n \leq 16$. The Y axis shows number of cycles, and the X axis shows number of active threads.

can be handled by an MMT-based hardware. All our benchmarks are highly parallel. To estimate TLP, we consider an ideal multi-threaded (MIMD) machine executing the instructions of each thread in sequence with a throughput and latency of 1 cycle, and with unlimited processing units. Figure 7 plots how often we had n, threads active at each execution cycle of the four PARSEC benchmarks, where $1 \leq n \leq 16$. Critical sections prevent us from having all the threads always active. Swaptions is the most "parallel" application. Its regularity makes it possible to have all the 16 available threads simultaneously active in 99.08% of all the execution cycles of the application. Bodytrack and Fluidanimate are less parallel, as we could expect from the number of instructions in critical sections shown in Figure 6. In Bodytrack, for instance, only the main thread is active in 44% of all the execution cycles. The average number of active threads, per cycle, for the Parsec benchmarks is: swaptions = 13.526, blacksholes = 14.835, bodytrack = 14.720 and fluidanimate = 4.812. The average number of active threads, per cycle, for the other benchmarks indicates a similar pattern: tachyon = 15.739, barnes = 5.322, fft = 15.164, fmm = 4.422, ocean_ncp = 15.785, radix = 14.463, volrend = 2.886 and water_nsquared = 11.534.

Microarchitecture-independant testbed: In order to compare thread

scheduling heuristics outside of a specific hardware implementation, we define an abstract execution model. In each cycle, this machine model can fetch one instruction, forward it to any number of threads and have each thread execute it. Inside each thread, instructions are issued in order. The execution order among threads is unconstrained outside of synchronization primitives.

5. Experimental Results

In this section we show results that we have obtained on scheduling policies using the methodology described in Section 4. We emphasize that the hardware in which we have performed these experiments has no influence in our observations, as they have been derived from simulations using Pin.

Throughput: Figure 8 compares the throughput of the thread synchronization heuristics that we have implemented. The throughput is the average number of threads that are active at each cycle. The longer the bar, the more efficient is the heuristic. For instance, in Figure 2(c), min-PC and min-SP/PC yield a throughput of 1.375. In that example, the maximum throughput would be 2.0, and the minimal, 1.0. Figure 8 shows that the min-SP/PC heuristic is more effective than the other alternatives. In this experiment we consider a maximum throughput of 16.0. Min-SP/PC gives us an average throughput of 9.799. Long's heuristic comes in second place, with an average throughput of 9.197. Surprisingly, we have observed that Min-PC, although much simpler, is slightly more effective on average than 2-stack. In our benchmark suite, the former approach gave us a throughput of 4.818, whereas 2-stack gave us 4.555.

5.1. Memory Access Patterns

Data locality is an important player in the development of high-performance programs. The literature traditionally considers two types of locality in sequential applications: spatial and temporal [15]. Data in close memory locations have good spatial locality. And data that is likely to be accessed often within short periods of time have good temporal locality. Recently, Meng *et al.* have introduced the notion of *inter-thread locality* in the context of the Single Instruction, Multiple Data (SIMD) execution model [16]. If two separate threads simultaneously read data from nearby memory cells, then these accesses are said to have good inter-thread locality. In this case, a single memory access might provide data to several different threads. The

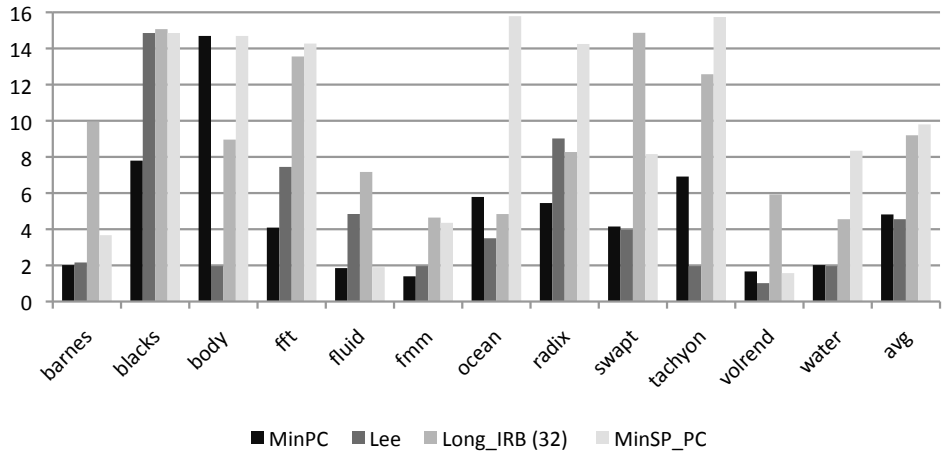

Figure 8: Throughput: average number of active threads per cycle. the longer the bar, the more effective is the heuristic.

importance of inter-thread locality is clear in the realm of graphics processing units, given that *memory access coalescing* is, according to many authors, the most important optimization in this environment [17, 18, 19].

In this paper we look into the potential of inter-thread locality in the context of minimal multi-threading. With this objective, we define three types of memory access patterns: uniform, strided and scattered. If we have n active threads executing a memory access instruction such as a load, e.g., $v = *a$, or a store, e.g., $*a = v$, then we assume that each thread i reads from, or writes to an address a_i. Given this assumption, we say that a memory access is uniform if $a_0 = a_1 = \ldots = a_n$. If $\mathtt{T}_{id}$ is an integer that uniquely identifies a thread, then we say that the access is strided if $a_i = c_1 \times \mathtt{T}_{id} + c_2$, for any two integer constants c_1 and c_2. Finally, if none of these patterns applies, then we say that the access is scattered.

Current multi-threaded hardware has not been designed to benefit from uniform and strided memory patterns: independent on the target address, n simultaneous threads require n accesses to memory ports. However, there exist proposals for new hardware designs that proceed differently [4]. In these processors, a uniform address causes only one access to the data cache. Likewise, if n threads execute a strided access, e.g., a load $v = *(c_1 \times \mathtt{T}_{id} + c_2)$, then a set of n memory cells, spaced by c_1 words, and starting at base address

c_2 is accessed once. In case c_1 is equal to the word size, accesses are contiguous and can be combined into a single memory transaction.

Local variables created during function calls, such as `t` in Figure 2, are placed in allocation units called *activation records*. These records are stored in a space called the *stack frame*. We want to ensure that whenever threads simultaneously access their private instance of the same local variable, we obtain a strided access. However, strided accesses to memory might be hidden by the relative disposition of the memory stack given to each thread. For instance, the Linux loader randomizes the base address of the stack given to different threads (ASLR) due to security reasons. We have disabled this randomization. In other words, we allocate the same amount of space for each thread to create its stack frame, and we make sure that the stack of activation records always starts at the beginning of this region. Hence, the local variables accessed by each thread are equally spaced.

Counting Access Patterns: Figure 9 shows how often each pattern is found in our simulations. We run tests for settings with 16 threads synchronized by the Min-SP/PC heuristic. We just show results for this heuristic because it has been the most effective to synchronize the threads, as we have observed in Figure 8. We only count patterns when we have the full number of threads active. It is not meaningful to distinguish strided from scattered access if we have just one thread in flight. In this case, we would classify every non-uniform memory access as scattered. Notice that we could just as well have classified them as strided. When we enable 16 threads, we find an encouraging amount of regularity in the memory access patterns of some of the benchmarks. For instance, we have observed 1.65 million memory accesses during the execution of bodytrack. Among these acceses, 32.23% are uniform, and 39.78% are strided. Swaptions gives us the largest number of accesses: 65.64 million, of which 77.12% use strided addresses.

Access Distance: If the data simultaneously accessed by active threads is within a short distance of each other, then it may fit into the same cache line. In this case, if the data is already cached, then every thread scores a hit. Otherwise, it can be brought to the cache with just one trip to a lower level in the memory hierarchy. Figure 10 shows the average maximum distance between addresses of strided accesses, considering the setting with 16 threads in flight. The actual interval between accesses, or stride, is one sixteenth of the access distance.

In general we have observed long distances between the addresses used by threads when simultaneously processing load and store instructions. The

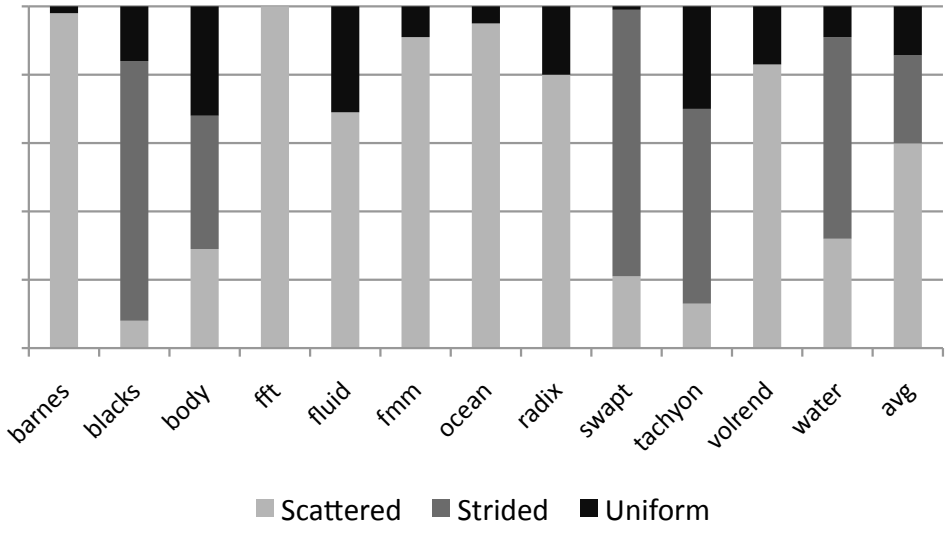

Figure 9: Access patterns with 16 active threads.

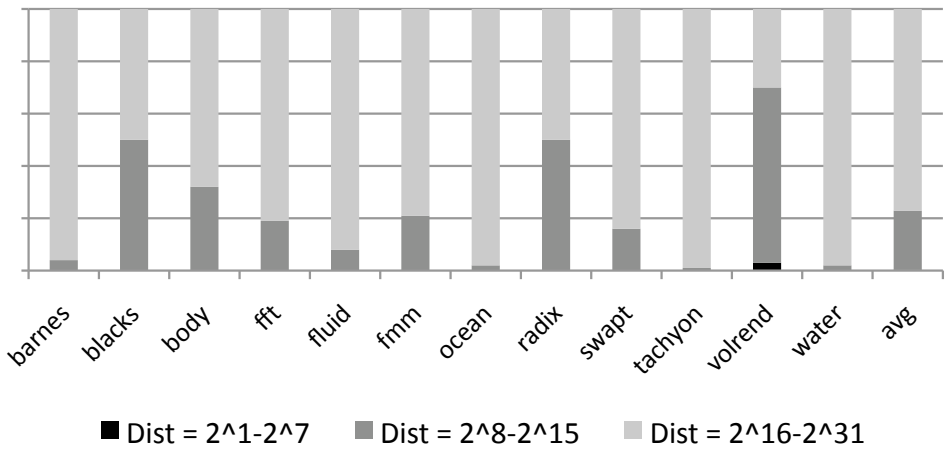

Figure 10: Maximum access distance among strided addresses.

longest observed distance among strided addresses is 2^{30}. Long spaces between strided accesses are common because each thread receives 2^{26} bytes of memory to allocate their stack frames. Thus, local variables stored in the

stack are likely to be spaced 2^{16} bytes. Closer addresses are found among strided accesses of data stored either in static memory, or, more usually, in the memory heap. Blackscholes and bodytrack use more these memory regions. During the execution of blackscholes, 43.98% of the memory accesses that happen are within a memory block less than 2^{15} bytes long. The proportion of these accesses in bodytrack is smaller: 36.92%. We have observed very close addresses only in volrend. In this benchmark, 2.75% of the memory accesses performed are within blocks less than 2^7 bytes long.

We speculate that these large spaces between addresses are common because the target benchmarks have not being coded with memory coalescing in mind. These benchmarks are meant to run in traditional CPUs, and in this case inter-thread locality is not at a premium. On the contrary, close inter-thread locality would be harmful in the context of multi-core platforms with coherent private caches, because it might cause false sharing of cache lines. Collange has observed a substantially different behavior in GPGPU applications [20]. In that case, inter-thread proximity is much more common, as this type of locality contributes notoriously to performance improvements.

6. Conclusion

In this paper, we have proposed a new thread synchronization heuristic: min-SP/PC. We have compared it against state-of-the-art algorithms on a microarchitecture-independent model. This empirical evaluation has demonstrated that min-SP/PC is very effective at keeping threads synchronized. We have also studied the memory access patterns typical of data-parallel multi-threaded applications, and have found a substantial amount of regularity between concurrent threads. This regularity is a further motivation for new hardware designs that have been proposed in the literature, but are yet to be manufactured. The paper brings in one negative result: data accessed by different threads tend to be distant in memory. This distance makes it difficult to take benefit from spatial locality in inter-thread memory accesses. It suggests that the data layout of the call stack should be reconsidered in the context of inter-thread locality. We leave this new study as future work.

References

[1] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, A. González, Thread fusion, in: Proceedings of the 13th international symposium on Low power electronics and design (ISLPED), ACM, 2008, pp. 363–368.

[2] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, F. T. Chong, Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors, in: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2010, pp. 337–348.

[3] M. Dechene, E. Forbes, E. Rotenberg, Multithreaded instruction sharing, Tech. rep., North Carolina State University (2010).

[4] B. W. Coon, J. E. Lindholm, System and method for managing divergent threads in SIMD architecture, US Patent 7353369 (2008).

[5] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, S. Yalamanchili, SIMD re-convergence at thread frontiers, in: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), ACM, 2011, pp. 477–488.

[6] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: characterization and architectural implications, in: Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT), ACM, 2008, pp. 72–81.

[7] J. E. Stone, An efficient library for parallel ray tracing and animation, Master's thesis, University of Missouri - Rolla (1971).

[8] F. Darema, D. A. George, V. A. Norton, G. F. Pfister, A single-program-multiple-data computational model for epex/fortran, Parallel Computing 7 (1) (1988) 11–24.

[9] M. J. Quinn, P. J. Hatcher, K. C. Jourdenais, Compiling C* programs for a hypercube multicomputer, SIGPLAN Not. 23 (1988) 57–65.

[10] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, K. Asanovic, Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators, in: International Symposium on Computer Architecture (ISCA), ACM, 2011, pp. 129–140.

[11] T. Milanez, C. Collange, F. M. Q. Pereira, W. J. Meira, R. Ferreira, Data and instruction uniformity in minimal multi-threading, in: Computer Architecture and High Performance Computing (SBAC-PAD), 2012, pp. 270–277.

[12] D. M. Tullsen, S. J. Eggers, H. M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, SIGARCH Comput. Archit. News 23 (1995) 392–403.

[13] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor, SIGARCH Comput. Archit. News 24 (1996) 191–202.

[14] P. Barone, P. Bonizzoni, G. D. Vedova, G. Mauri, An approximation algorithm for the shortest common supersequence problem: an experimental analysis, in: ACM symposium on Applied computing (SAC), ACM, 2001, pp. 56–60.

[15] J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, 3rd Edition, Elsevier, 2003.

[16] J. Meng, D. Tarjan, K. Skadron, Dynamic warp subdivision for integrated branch and memory divergence tolerance, in: International Symposium on Computer Architecture (ISCA), ACM, 2010, pp. 235–246.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W. mei W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Symposium on Principles and practice of parallel programming (PPoPP), ACM, 2008, pp. 73–82.

[18] Y. Yang, P. Xiang, J. Kong, H. Zhou, A GPGPU compiler for memory optimization and parallelism management, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 86–97.

[19] A. Lashgar, A. Baniasadi, Performance in GPU architectures: Potentials and distances, in: 9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD 2011), IEEE, 2011, pp. 75–81.

[20] C. Collange, D. Defour, Y. Zhang, Dynamic detection of uniform and affine vectors in GPGPU computations, in: Euro-Par 2009 – Parallel Processing Workshops, Springer, 2009, pp. 46–55.