Data and Instruction Uniformity in Minimal Multi-Threading

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

Data and Instruction Uniformity in Minimal Multi-Threading *

Teo Milanez Caroline Collange Fernando Magno Quintão Pereira
Wagner Meira Jr. Renato A. Ferreira
Departamento de Ciência da Computação Universidade Federal de Minas Gerais, Brazil
{milanez,sylvain.collange,fernando,meira,renato}@dcc.ufmg.br

September 12, 2012

Abstract

Simultaneous Multi-Threading (SMT) is a hardware model in which different threads share the same instruction fetching unit. This model is a compromise between high parallelism and low hardware cost. Minimal Multi-Threading (MMT) is a technique recently proposed to share instructions and execution between threads in a SMT machine. In this paper we propose new ways to explore redundancies in the MMT execution model. First, we propose and evaluate a new thread reconvergence heuristics that handles function calls better than previous approaches. Second, we demonstrate the existence of substantial regularity in inter-thread memory access patterns. We validate our results on the four data-parallel applications present in the PARSEC benchmark suite. The new thread reconvergence heuristics is, on the average, 82% more efficient than MMT's original reconvergence method. Furthermore, about 69% to 87% of all the memory addresses 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.

1 Introduction

Resource sharing at the hardware level is an alternative that computer architects have been adopting to decrease the costs of highly parallel processors. One form of resource sharing is known as *minimal multi-threading* [1], or MMT for short. An MMT-based architecture organizes threads into 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 all of them receive an instruction to execute. If this instruction has the same input values, then the computations are combined as well, 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, Long *et al.*'s original formulation uses an intricate reconvergence heuristics, which, in the words of the authors themselves, has impact on the hardware's performance [1]. This heuristics is expensive because it looks up the program counter's history every execution cycle. On the other hand, the previous alternative, Quinn's algorithm, which exists since the late 80's [2], cannot couple 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. 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 [3].

The objective of this paper is to advance the research on minimal multi-threading, a task that we accomplish in two ways. First, we propose a new heuristics to keep threads synchronized. 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 the four data-parallel applications found in the PARSEC benchmark suite [4]. We analyze only the data-parallel ap-

*This work was partially supported by CNPq, CAPES, FAPEMIG and InWEB.

plications because in this case threads execute the same program, following Darema's SPMD model [5]. Hence, we have more opportunities to share instructions among threads. These experiments are meaningful because the PARSEC programs are remarkably large and complex, and they give us execution traces with billions of x86 instructions.

Our first contribution is a new thread reconvergence heuristics that improves on Quinn's algorithm [2]. Keeping threads as much synchronized as possible is important because 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. Quinn's reconvergence criterion gives priority to the thread with the minimum program counter. This strategy was proposed to reconverge SIMD programs on distributed computers, and today it is used to synchronize threads in GPU-like architectures [6, 7]. However, it requires the compiler and linker to statically lay out the binary code of each function according to the function call graph, complicating the build process and preventing its use on existing applications. Our new heuristics gives priority to the thread with the minimum stack-pointer. In case of ties, it then uses the minimum PC criterion. We call it min-SP/PC. Our experiments show that the min-SP/PC heuristics is 82% more effective than min-PC.

Our second contribution is an analysis of the *memory access patterns* in the MMT setting. Such patterns describe the relative arrangement of addresses in the load and store instructions used by each thread. We have identified three different access patterns: *uniform*, *affine* and *scattered*, which we shall define in Section 4. As we show in that section, we have observed substantial regularity in inter-thread access patterns. These forms of regularity have not been previously noticed because Long *et al.* [1] 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 [3]. For instance, if all the threads read data from the same location, or from regularly spaced locations, then the hardware can bring all this data to registers with only one cache access. Patterns in this fashion happen in over 70% of all the memory accesses. On the other hand, if simultaneous memory accesses are randomly scattered, then we lose inter-thread locality, and access fragmentation puts an increased pressure on caches.

MMT in Perspective: Instruction and data sharing are not new ideas in the computer architecture world. In the mid nineties Tullsen *et al.* introduced the notion of *Simultaneous Multi-Threading* (SMT) [8, 9]. 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. In 2008 Gonzalez *et al.* brought in the concept of *Thread Fusion*, as a way to decrease the energy consumption of SMT machines [10]. Thread fusion consists in giving 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. Long *et al.* [1] extended Gonzalez's work by introducing the idea of minimal multi-threading. They have designed a hardware mechanism that reconverges threads without the intervention of a compiler. Notice that 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.

2 Methodology

The Benchmarks: In this paper we chose to analyze the four data-parallel applications present in the PARSEC benchmark suite:

- **Blackscholes:** option pricing with Black-Scholes Partial Differential Equation (PDE).
- **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.
- **Swaptions:** application that uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions.

These four programs are data-parallel applications implemented in C, on top of the *pthreads* library. Figure 1 shows some characteristics of these applications.

Benchmark	S	C	D
Blackscholes	27,025	0	276,503,954
Bodytrack	71,597	62,682	1,136,165,901
Fluidanimate	32,812	6,291	6,375,221,686
Swaptions	31,246	0	1,972,213,570

Figure 1: Characteristics of the benchmarks. (S) Number of x86 instructions in the program text. (C) Instructions in critical section executed by a single thread. (D) Number of instructions executed in a single-threaded processor with *simsmall* input.

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 **simsmall** input [4, p.73].

Simulation: We have instrumented the binary programs using the *Pin* framework[1]. 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 can be handled by an MMT-based hardware. The four applications that we have chosen are highly parallel. To estimate TLP, we consider an ideal multi-threaded machine executing the instructions of each thread in sequence with a throughput and latency of 1 cycle, and with unlimited TLP. Figure 2 plots how often we had $n, 1 \leq n \leq 16$ threads active at each execution cycle. 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 1. In Bodytrack, for instance, only the main thread is active in 44% of all the execution cycles.

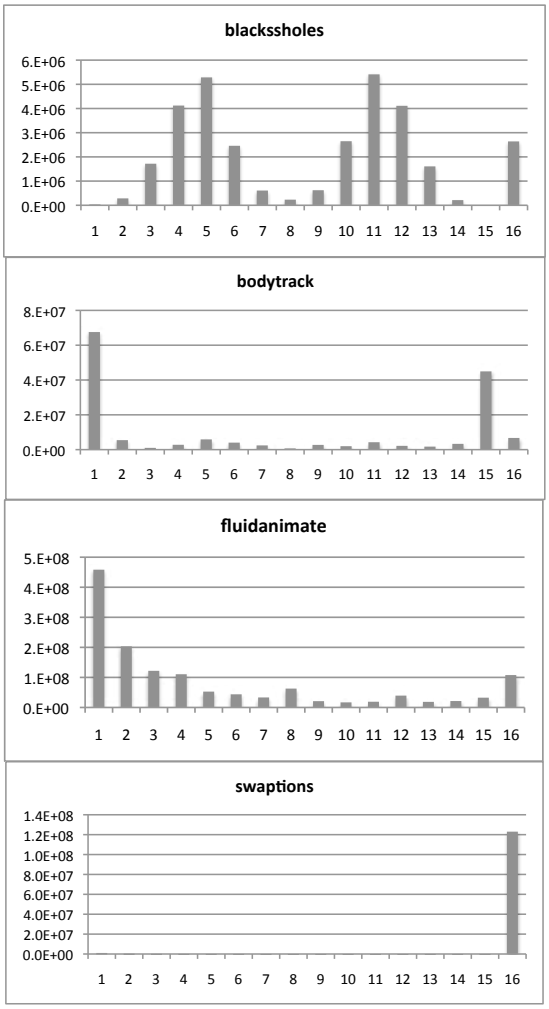

Figure 2: Histogram showing the number of cycles in which n threads were active, where $1 \leq n \leq 16$.

3 Instruction Sharing

We are interested in maximizing the amount of common instructions executed by independent threads. In other words, we want to keep these threads as much synchronized as possible. We define the maximal sharing problem as follows:

Definition 3.1 MAXIMAL 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*

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

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*. 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* problem, and it is NP-complete [11]. However, we are more interested in the on-line version of maximal sharing. In this case, each trace t is seen as a *stack*, with top at $t[0]$. We can only perform one of two operations on each t_i: "inspect top t_i" and "pop t_i". The first operation lets us see the opcode at the top of t_i. The second removes the opcode from the top of t_i, and places it at the top of T. Notice that we are not allowed to store opcodes before inserting them into T. Due to this last restriction, plus the fact that traces might contain random sequences of opcodes, there is no algorithm that solves on-line maximal sharing optimally [12]. Therefore, this problem must be solved by heuristics.

3.1 Heuristics for Instruction Sharing

While simulating the PARSEC programs, we have experimented with two different instruction schedulers, which were based on the the min-PC and the min-SP/PC heuristics. Before presenting numbers, it is interesting to discuss why threads diverge, and how each heuristics reconverges them. Figure 3 illustrates the min-PC heuristics. We assume that the hardware supports two threads executing simultaneously the same instruction. The program in Figure 3(a) is written in a C-like language with a special keyword, **fork**, that passes a function to a freshly created thread. Figure 3(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 heuristics keeps feeding the thread that is reading instructions from the lowest program counter. The rational behind this heuristics is that the next instruction that is not controlled by a branch is normally located below that branch in the program code. In this example shared instruction fetching resumes at instruction six.

Unfortunately, the min-PC heuristics might take too long to reconverge threads in code that contains function calls. Figure 4 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 3(c).

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 *downward-growing* stack, 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. This heuristics provides optimal reconvergence in the example of Figure 4. We show empirically that this heuristics fits well the existing SPMD applications expected in the minimal multi-threading scenario.

Figure 5 compares these two heuristics. The measure of efficiency, in this case, is the total number of instructions executed by an MMT-enabled hardware. The shorter the length of the instruction trace, the more efficient is the heuristics. For instance, in Figure 4(c), the min-PC heuristics produces a trace with ten instructions. On the other hand, the min-SP/PC heuristics would produce a trace with eight instructions, because it would be able to share program counters 4 and 5. As we can see in the charts of Figure 5, the min-SP/PC heuristics is more efficient. On the average, it produces traces 33.3% shorter for blackscholes, 33.8% shorter for bodytrack, 33.1% shorter for fluidanimate and 46.7% shorter for swaptions, given two threads in flight. With 16 threads the difference is even larger: 74.1%, 74.4%, 75.3% and 74.4% respectively. The min-SP/PC heuristics produces remarkably good results for very regular applications. For example, for swaptions, over 98% of all the instructions issued with 16 threads are shared among all the threads.

4 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 [13]. Data

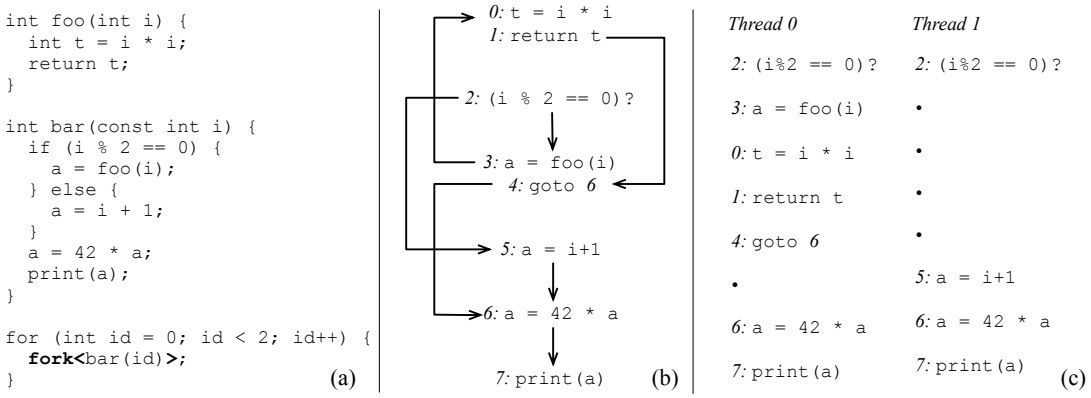

Figure 3: A example where the min-PC heuristics is able to reconverge optimally divergent threads.

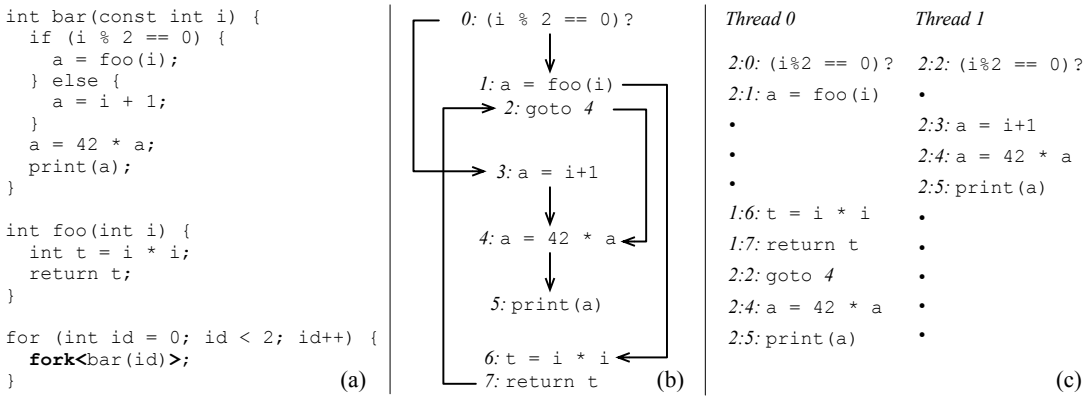

Figure 4: Example where the min-PC criterion fails to reconverge divergence threads, but the min-SP/PC does it optimally. In part (c) we prefix each instruction with its stack pointer plus program counter.

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 [14]. 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 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 [15, 16, 17].

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, affine 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 address is affine 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 de-

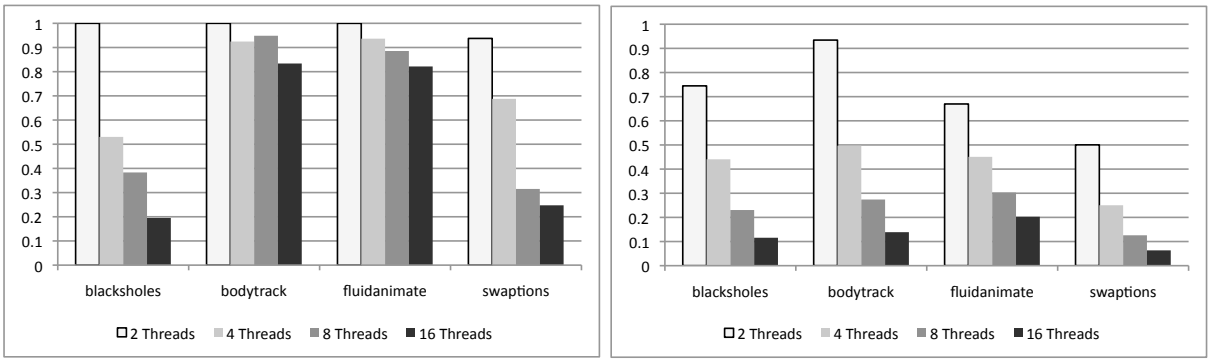

Figure 5: (Left) Instruction trace reduction produced by the min-PC heuristics. (Right) Instruction trace reduction produced by the min-SP/PC heuristics.

signed to benefit from uniform and affine 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 [3]. In these processors a uniform address causes only one access to the data cache. Likewise, if n threads execute an affine 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 at once. In case c_1 is equal to the word size, accesses are contiguous and can be combined into a single memory transaction.

Counting Access Patterns. Figure 6 shows how often each pattern is found in our simulation of the PARSEC data-parallel applications. We run tests for settings with 2 and 16 threads, and we only count patterns when we have the full number of threads active. It is only meaningful to distinguish affine from scattered access if we have more than two threads in flight. Otherwise we classify every non-uniform memory access as scattered. Notice that we could just as well have classified them as affine. When we enable 16 threads, we find an encouraging amount of regularity in the memory access patterns. For instance, we have observed 1.65 million memory accesses during the execution of bodytrack. 44.23% of these accesses are uniform, and 30.78% are affine. Swaptions gives us the largest number of accesses: 65.64 million, of which 73.12% use affine addresses.

The large quantity of affine accesses that we have observed is mostly due to the fact that we use a customized loader to allocate stack frames. Local variables created during function calls, such as `t` in Figure 3, are placed in allocation units called *activation records*. These records are stored in a space called the *stack frame*. The core idea of the loader is very simple: 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. In this way, the local variables accessed by each thread are equally spaced.

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 7 shows the average maximum distance between the data accessed by all the threads, considering the setting with 16 threads in flight. Each number, e.g., 0 to 63, is the base-2 logarithm of the maximum distance between any two addresses given sixteen simultaneous memory accesses.

In general we have observed very long distances between the addresses used by threads when simultaneously processing load and store instructions. Figure 7(Left) shows the maximum distances between affine addresses. The longest distance that we have observed in this case is 2^{30}. Long spaces between affine 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^{26} bytes. Closer addresses are found in affine accesses of data stored either in static memory, or, more usually, in the memory heap. Blackscholes and bodytrack use more these memory

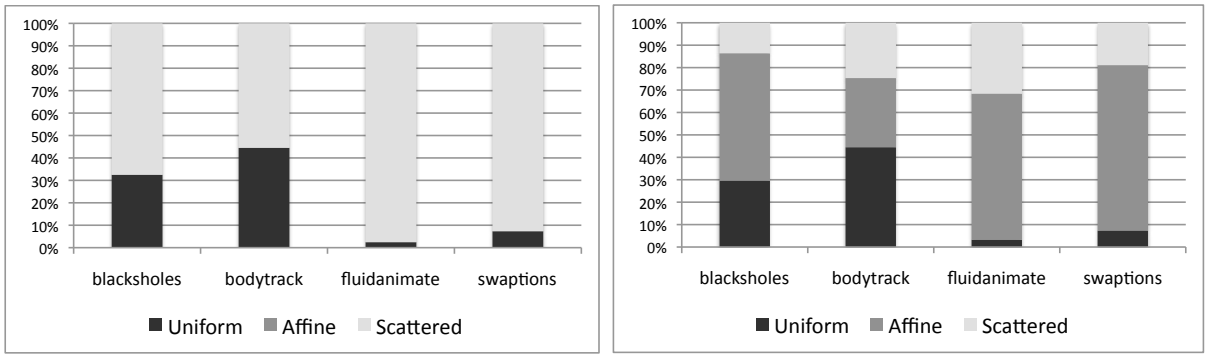

Figure 6: (Left) Access patterns with two active threads. (Right) Access patterns with 16 active threads.

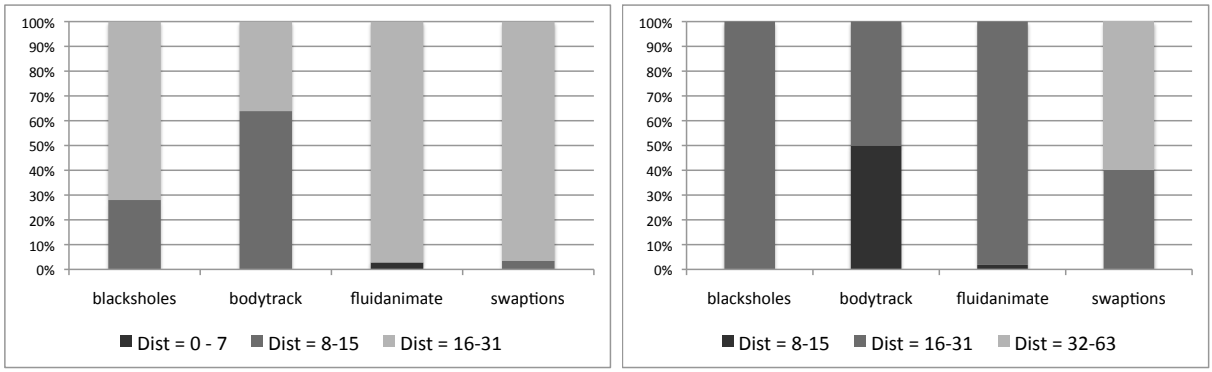

Figure 7: (Left) Maximum access distance among affine addresses. (Right) Maximum access distance among scattered addresses.

regions. 27.98% of the memory accesses that happen during the execution of blackscholes are within a memory block less than 2^{15} bytes long. The proportion of these accesses in bodytrack is even larger: 63.92%. We have observed very close addresses only in fluidanimate. 2.75% of the memory accesses performed by this benchmark are within blocks less than 2^7 bytes long. Figure 7(Right) shows the maximum distance among scattered accesses. In this case, we have observed accesses distant as much as 2^{40} bytes away. Furthermore, we have not observed very close accesses, e.g., within blocks up to 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. The PARSEC 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, by causing false sharing of cache lines. Collange has observed a substantially different behavior in GPGPU applications [18]. In that case, inter-thread proximity is much more common, as this type of locality contributes notoriously to performance improvements.

5 Conclusion

In this paper we have advanced the research on minimal multi-threaded hardware in two different directions. First, we have proposed a new thread synchronization heuristics: min-SP/PC, and compared it with min-PC, a heuristics originally proposed

for SIMD languages. Our empirical evaluation has demonstrated that min-SP/PC is remarkably more efficient to keep 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 the data layout of call stack memory should be reconsidered in the context of inter-thread locality. We leave this new study as future work.

References

[1] G. Long, D. Franklin, S. Biswas, P. Ortiz, J. Oberg, D. Fan, and F. T. Chong, "Minimal multi-threading: Finding and removing redundant instructions in multi-threaded processors," in *MICRO*. IEEE, 2010, pp. 337–348.

[2] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais, "Compiling C* programs for a hypercube multicomputer," *SIGPLAN Not.*, vol. 23, pp. 57–65, 1988.

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

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *PACT*. ACM, 2008, pp. 72–81.

[5] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for epex/fortran," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.

[6] G. Diamos, A. Kerr, H. Wu, S. Yalamanchili, B. Ashbaugh, and S. Maiyuran, "SIMD reconvergence at thread frontiers," in *MICRO*, 2011.

[7] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanovic, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *ISCA*. ACM, 2011, pp. 129–140.

[8] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," *SIGARCH Comput. Archit. News*, vol. 23, pp. 392–403, May 1995.

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

[10] J. González, Q. Cai, P. Chaparro, G. Magklis, R. Rakvic, and A. González, "Thread fusion," in *ISLPED*. ACM, 2008, pp. 363–368.

[11] P. Barone, P. Bonizzoni, G. D. Vedova, and G. Mauri, "An approximation algorithm for the shortest common supersequence problem: an experimental analysis," in *SAC*. ACM, 2001, pp. 56–60.

[12] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. Cambridge University Press, 1998.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Elsevier, 2003.

[14] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *ISCA*. ACM, 2010, pp. 235–246.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *PPoPP*. ACM, 2008, pp. 73–82.

[16] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *PLDI*. ACM, 2010, pp. 86–97.

[17] A. Lashgar and A. Baniasadi, "Performance in GPU architectures: Potentials and distances," in *WDDD*. IEEE, 2011, pp. 75–81.

[18] C. Collange, D. Defour, and Y. Zhang, "Dynamic detection of uniform and affine vectors in GPGPU computations," in *HPPC*. Springer, 2009, pp. 46–55.