smt-*SPRINTS*: *S*oftware *Pr*ecomputation with *Int*elligent Streaming for Resource-Constrained SMTs

Tanping Wang, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos

Department of Computer Science
The College of William and Mary
McGlothlin–Street Hall. Williamsburg, VA 23187–8795
{twang,cda,dsn}@cs.wm.edu

Abstract. We present *SPRINTS*, a source-level speculative precomputation framework for scientific applications running on SMTs with two execution contexts. Our framework targets memory-bound applications and reduces memory latency by prefetching long streams of delinquent data accesses. A unique aspect of *SPRINTS* is that it requires neither hardware nor compiler support. It is based on partial cache simulation and a compression algorithm which can accurately summarize very long streams of cache misses. *SPRINTS* extracts patterns from the streams, which are in turn used to generate source-level, highly optimized precomputation code. *SPRINTS* achieves significant performance improvements over plain thread-level parallelization and indiscriminate precomputation based on code cloning. We demonstrate these improvements using two realistic scientific applications.

1 Introduction

Simultaneous multithreading (SMT) allows multiple threads to concurrently issue instructions to different execution units of the same physical processor. SMT has been recently used as a core architecture by several processor manufacturers [7, 11, 14], since it has the potential of achieving better performance than conventional superscalar processors, at a minimal additional cost. The main reason for the cost effectiveness of SMT processors is that threads share a common set of execution resources. The major shortcoming of resource sharing is that it may result to performance loss, should threads on the processor end up competing for resources such as execution units, instruction buffers and cache space. This performance loss is most noticeable in parallel scientific computations, in which threads tend to be memory-bound and to have identical resource requirements.

Speculative precomputation (SPR) [4] is a technique which uses thread contexts in an SMT to eliminate L2 cache misses from the main computation threads, by pre-executing future memory accesses. SPR has demonstrated the potential of speeding up pointer-based, single-threaded code on multithreaded processors and several hardware and software implementations have been investigated in the related literature [3, 8–10, 12]. This paper makes a case for using SPR as an alternative to thread-level parallel execution on SMTs with two hardware contexts and limited execution resources. The motivation for using SPR in scientific codes stems from two observations. First, the

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 710–719, 2005.
© Springer-Verlag Berlin Heidelberg 2005

hardware of existing SMTs can not handle the resource pressure from multiple memory- and execution unit-bound threads. A carefully designed SPR scheme can reduce this pressure to a minimum, while still reducing memory latency suffered by co-executing application code. Second, most memory-bound scientific codes suffer from memory latency caused by long, but quite predictable streams of memory accesses. SPR is a mechanism which can effectively prefetch such streams.

The contribution of this paper is a user-level software SPR framework, which supports stream-based SPR for scientific applications, with no hardware or compiler support. We named our framework *SPRINTS* (*S*oftware *PR*ecomputation with *INT*elligent *S*treaming). In the heart of this framework lies a compression and pattern extraction algorithm, the purpose of which is to identify streams of delinquent loads which can be directly mapped to streams of data accesses in the source code of the program. Although multiple forms of streams exist in a program (such as dynamic instruction streams, streams of data addresses and so on), our framework opts for identifying a form of streams which can be directly mapped back to computation and data structures in the source code. *SPRINTS* represents streams of L2 cache misses as strings of inter-miss iteration distances, using feedback from a cache simulator. It uses the compression grammar to identify strong patterns in the loop iterations that incur L2 cache misses and feeds these patterns back to a source code generator. The source code generator translates streams into precomputation loops which have small instruction working sets and are amenable to optimization by the back-end compiler.

SPRINTS has a number of advantages, both as a self-contained tool and compared to other precomputation strategies. The first is simplicity, as there is no requirement for compile-time analysis, or additional hardware to trace the code. The second is automation and transparency to the programmer. The third is portability across SMT architectures. The stream identification and compression/decompression engine is independent of the target architecture and works end-to-end using only source code. Finally, *SPRINTS* is engineered for high performance, using optimizations such as store removal, prefetch distance control, and prefetch target selection. We have evaluated *SPRINTS* on a Hyperthreaded Intel Xeon. Our results show that *SPRINTS* speeds up scientific applications for which thread-level parallelization performs poorly.

The rest of the paper is organized as follows: Section 2 introduces *SPRINTS* and provides implementation details. The experimental evaluation of *SPRINTS* is presented in section 3. In section 4 we discuss related work. Finally, section 5 concludes the paper.

2 Precomputation with Intelligent Streaming

SPR implementations [4, 15] adopt a "top-10" approach for identifying delinquent loads and emitting precomputation code. More specifically, architectural simulation is used to identify a few loads which are responsible for most cache misses. Once these loads are identified, code cloning and slicing are performed to issue the instruction paths that lead to the delinquent loads and the loads themselves in the precomputation thread. The "top-10" approach works well in practice because in most codes a few static loads are responsible for large fractions (e.g. more than 90%) of cache misses. Another reason for using this approach is that precomputation threads interfere with their sibling

computation threads, sharing execution units and other resources in the processor. Indiscriminate code cloning in precomputation threads would cause excessive interference, whereas highly selective code cloning and further optimization of the precomputation code will reduce the interference. For these reasons we adopt the "top-10" approach for precomputation in *SPRINTS*.

SPRINTS uses cache simulation to identify delinquent loads. The practical impact of using simulation is that *SPRINTS* can accurately identify critical memory accesses that need to be prefetched. In contrast, mere code profiling would only discover dominant reference streams without necessarily revealing information on the cache behavior of these streams. Cache simulation is a generic method which makes *SPRINTS* portable with reasonable effort between SMT architectures. Porting *SPRINTS* requires porting of the cache simulator to accept memory reference traces from a different ISA and adaptation of the architectural parameters of the targeted processor's cache. The porting process can be facilitated by several existing simulation tools.

The profiling mechanism of *SPRINTS* has two distinctive features. Besides detecting delinquent loads, it also recognizes repetitive patterns in these loads. Moreover, it maps delinquent loads identified via profiling back to source code and actually emits source code (at the language level) in precomputation threads. This code prefetches directly elements of application-level data structures. Currently we map misses back to array elements, but the same tool can be used to map back to elements of other data structures as well. *SPRINTS* uses partial simulation, taking advantage of the iterative structure of scientific codes to simulate only a few outermost iterations of the dominant loops and save significantly on simulation time.

SPRINTS targets loop-based scientific applications, which exhibit strong patterns in several aspects of their control and data flow. In particular, the loop-intensive structure of scientific applications, and the fact that delinquent loads tend to occur in heavily traversed loops [8] motivate the use of a loop-based approach to precomputation, in which the speculative thread prefetches *streams* of data that would otherwise be streams of cache misses. *SPRINTS* uses a grammar which detects such streams, by tracking the loop iterations in which delinquent loads occur and identifying patterns of distances between delinquent loads, measured in loop iterations. The rationale for this technique is that long streams of loop iterations with delinquent loads, when mapped back to source code, can be directly translated to highly optimizable source code loops. Furthermore, using loops for precomputation allows *SPRINTS* to trigger precomputation in synch with the sibling computation threads, using loop levels as natural synchronization boundaries and specific loop iterations as natural trigger points. This property is desirable because it allows for accurate and effective control of the *runahead distance* between precomputation and sibling computation, which is in turn critical for timely prefetching [8, 15]. The following sections outline the main components of *SPRINTS*.

2.1 Cache Simulation and Trace Collection

SPRINTS uses a cache simulator based on Cachegrind, the cache simulation component of Valgrind[1], to obtain complete traces of cache misses. We have modified Cachegrind

[1] http://developer.kde.org/~sewardj/docs-2.2.0/manual.html

to analyze the instruction address stream and detect backward arcs in the dynamic control flow graph of the program. Backward arcs may correspond to loops, however they may also correspond to other control structures. *SPRINTS* uses *objdump*, a GNU development tool, to uniquely identify loops in the program. The tool is used to disassemble the object file and extract the first instruction address of the body of each loop. The simulator uses these instructions as anchors in order to both correctly identify loops and keep track of the loop iteration count. We have introduced a new module in Cachegrind to map uniquely memory references that miss in the cache to array elements, using the current loop iteration count as input.

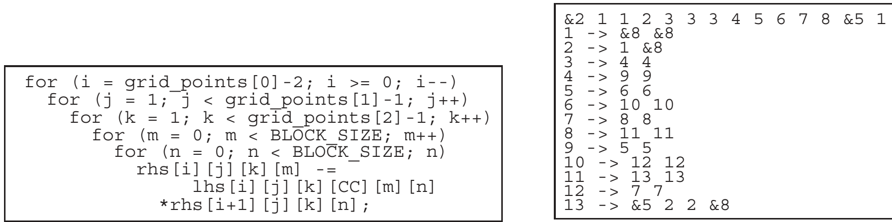
2.2 Delinquent Load Identification and SPR Code Generation

Following profiling, *SPRINTS* reorganizes the trace of cache misses, and groups misses by associating them with the addresses of the instructions that trigger them. Typically, few static instructions are responsible for most misses and misses from the same instruction tend to exhibit strong repetitive patterns.

SPRINTS uses the Sequitur [1] grammar to compress the trace of misses into a compact representation. The representation stores the misses as strings of symbols, with each string corresponding to one array reference accessed in one loop nest. Each symbol in a string represents the distance (in loop iterations, measured after loop linearization) from the previous miss for the same data object in the same nest. The grammar is composed of rules in which terminals (symbols) represent unique distances between consecutive misses and non-terminals represent concatenations of terminals which uniquely identify the stream of cache misses for each array reference. Sequitur constructs a context-free grammar with exactly one word for each reference. The grammar can be represented as a set of DAGs and the whole stream of misses can be reproduced (uncompressed) from the grammar with one preorder pass, in time linear to the length of the grammar. The length of each string is equal to the number of misses incurred by the corresponding reference, which can then be quantified as a fraction of the total number of cache misses incurred in the whole program and used to classify the reference as delinquent or not. In order to identify strong patterns with Sequitur, it suffices to find non-terminals (sub-strings of the grammar) with multiple occurrences. Each such substring can be translated to a loop which prefetches the stream. These loops are highly optimizable[2] and even parallelizable from a standard back-end compiler.

Consider Figure 1, which shows a loop from the NAS BT benchmark. The loop belongs to the `x_backsubstitute` function of the `x_solve` module of BT. The right part of the figure shows the rules of the Sequitur grammar which describes all the cache misses incurred in accesses to elements of `lhs`, in a single string of inter-miss loop iteration distances. For further details on how this grammar is constructed the reader is referred to [1]. In the illustrated example, all integers prefixed with an ampersand are terminal symbols and all other symbols are non-terminals. One can easily observe that after the first miss, cache misses exhibit a very strong pattern with inter-miss distances

[2] Dead-code elimination is the only optimization that needs to be precluded in precomputation loops. Furthermore, *SPRINTS* replaces delinquent stores with loads to preserve correctness in the architectural state.



```
for (i = grid_points[0]-2; i >= 0; i--)
  for (j = 1; j < grid_points[1]-1; j++)
    for (k = 1; k < grid_points[2]-1; k++)
      for (m = 0; m < BLOCK_SIZE; m++)
        for (n = 0; n < BLOCK_SIZE; n)
          rhs[i][j][k][m] -=
               lhs[i][j][k][CC][m][n]
            *rhs[i+1][j][k][n];
```

```
&2 1 1 2 3 3 3 4 5 6 7 8 &5 1
1 -> &8 &8
2 -> 1 &8
3 -> 4 4
4 -> 9 9
5 -> 6 6
6 -> 10 10
7 -> 8 8
8 -> 11 11
9 -> 5 5
10 -> 12 12
11 -> 13 13
12 -> 7 7
13 -> &5 2 2 &8
```

Fig. 1. Sample loop of `x_backsubstitute` in NAS BT and compression grammar for the cache misses incurred by elements of `lhs`, during execution with the Class A problem size.

predominantly equal to 8 iterations, and sporadically equal to 5 iterations. The grammar given in this example describes a total of 6 million cache misses on element `lhs` (spread over 200 iterations executed by BT, with approximately 30 thousand misses each) with only 13 rules and a couple of hundreds bytes of storage. A back of the envelope calculation will show that the entire cache miss sequence of the specific data access is represented uniquely with the string: $28^7 A^{3072} A^{512} A^{128} A^{64} A^8 A^4 58^2$, where $A = 58^7$. The grammar is easily translated into tight loops for prefetching the cache-missing elements of `lhs` using a recursive algorithm which visits each rule of the grammar in order.

The precomputation code generation phase of *SPRINTS* uses the loop iterations as natural units for controlling the distance between the precomputation and sibling computation threads. Furthermore, it uses loop iterations to throttle the precomputation thread, so that the data fetched in a stream do not overflow the L2 cache. Both techniques (runahead distance control and throttling) derive from our earlier work [15]. Another optimization applied by *SPRINTS* is the release of processor resources held by a precomputation thread when the latter is idling and not fetching streams.

3 Experimental Evaluation

We present experiments obtained with the OpenMP, C versions of BT and FT, two realistic application codes from the NAS benchmarks suite [6], both using the class A problem size. BT is a simulated CFD application which uses an implicit finite-difference algorithm based on the alternate direction implicit method, to solve 3-dimensional compressible Navier-Stokes equations. FT implements a solver for a class of PDEs using a 3-dimensional bidirectional (forward and inverse) complex FFT. BT and in are good candidates for speculative precomputation techniques, because their parallelized versions exhibit performance degradation (in the case of BT), or very modest performance gains (in the case of FT), when executed on SMTs with two execution contexts. The performance bottlenecks of parallelization stems from contention for execution units and cache space. A speculative precomputation thread can alleviate these problems and provide speedup by reducing memory latency. The applications have been compiled with the Intel C/C++ OpenMP compiler, using the highest level of optimization. Our hardware platform is a four-way SMP with Intel's Hyperthreaded Xeon processors, clocked at 1.4 GHz. Each processor offers two execution contexts and is equipped with 8KB L1 data cache, 12KB L1 instruction trace cache and 256 KB unified L2 cache.

The Hyperthreaded processors include a hardwired hardware prefetching engine, which was active throughout all the experiments. The hardware prefetching engine may interfere with software prefetcing engines, such as *SPRINTS*, by detecting and prefetching some of the references prefetched also by the software prefetching engine. This effect can not be quantified with the tools available on the specific processor. It must be noted that the automatic software prefetching engine of the Intel compiler was activated in the baseline sequential execution of the benchmarks, as well as in parallel executions of the benchmarks with two execution contexts per processor. However, the Intel's prefetching engine was deactivated while generating code with *SPRINTS*. We have also experimented with manual, non-speculative software prefetching via directives to the Intel compiler in both single-threaded and multithreaded versions of the codes, but we have not seen appreciable performance improvements. In the experiments with *SPRINTS*, we have used a runahead distance of one iteration for each loop targeted by the software precomputation engine. The distance was controlled without synchronization, by having the precomputation thread prefetch references from the second iteration onwards.

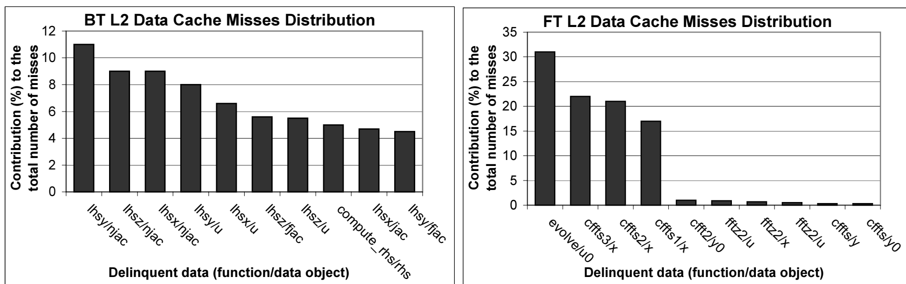


Fig. 2. The top delinquent data objects and their contribution to the total number of L2 data cache misses for BT (left diagram) and FT (right diagram).

Figure 2 depicts the contribution of the top 10 delinquent data accesses of BT and FT, to the total number of L2 data cache misses for the two applications. Those objects are responsible for 85% and 91% of the total cache misses in BT and FT respectively. In fact in FT, 4 objects generate 85.5% of the cache misses. It is thus reasonable to generate precomputation code targeting just the top few delinquent objects.

Following, we evaluate the impact of 4 different execution strategies to the number of L2 data cache misses suffered by the applications. The results are depicted in figure 3. *ST* stands for the single-threaded execution with one execution context in the processor. In *TLP* (*T*hread *L*evel *P*arallelism) mode, applications are executed in parallel by two threads, each one on a different execution context of the processor. The *SPR* (*S*peculative *PR*ecomputation) scheme exploits one of the contexts to execute a precomputation thread, which indiscriminately preexecutes all the memory references of the computation thread in each loop nest where *SPR* is applied. Finally, *SPRINTS* stands for the execution of the application using our precomputation framework. The precomputation thread of *SPR* executes exactly the same loops as *SPRINTS*.

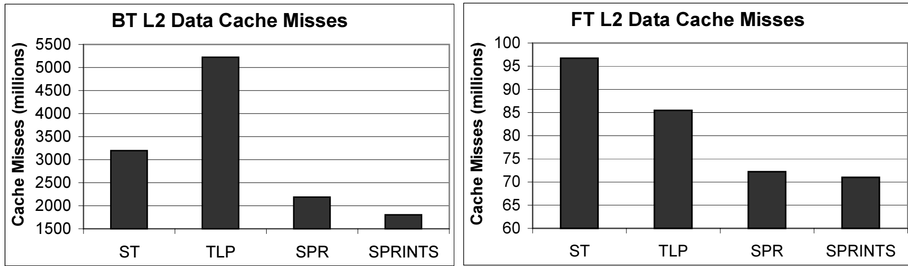


Fig. 3. L2 data cache misses under the four different execution strategies for BT and FT.

As expected, both *SPR* and *SPRINTS* significantly reduce the number of L2 data cache misses. *SPR* results to 31.6% and 25.3% less misses for BT and FT respectively. The corresponding percentages for *SPRINTS* are 42.7% and 25.6%. Although the data accesses targeted by the precomputation strategies are responsible for 85% and 91% of the misses triggered by BT and FT respectively, none of the strategies is successful in eliminating all the misses. Moreover, despite the fact that they both target the same loops and *SPR* touches more data than *SPRINTS*, *SPRINTS* outperforms *SPR* in all cases. This difference can be explained by a closer look at the characteristics of the benchmarks. Both BT and FT have tight, memory intensive loops. As a result, the execution time of the precomputation and computation loop bodies is comparable, since the precomputation thread suffers the latency of cache misses and is as much memory-bound as the computation thread. This means that computation may run side-to-side or even overrun precomputation, reducing the effectiveness of the latter. Since *SPRINTS* produces more compact precomputation code than *SPR*, this adverse behavior occurs less often and the miss coverage is better.

The effect of *TLP* on cache performance is also highly dependent on the characteristics of applications. The two threads of BT contend for L2 cache space, since their working sets do not fit in the cache. This results to a dramatic increase of 63% in cache misses. Contrary to BT, the threads of FT have smaller working sets that fit in the L2 cache. Moreover, they share data and each thread benefits from data prefetched to the cache by the other thread. As a consequence, the multithreaded execution suffers less L2 data cache misses than the sequential execution.

Table 1. Speedups over the single threaded execution using the alternative execution strategies.

	TLP	*SPR*	*SPRINTS*
BT	0.76	1.02	1.08
FT	1.03	1.03	1.05

Table 1 shows the speedups achieved by the three execution strategies which exploit both execution contexts of the processor over the single-threaded execution. The performance of the *TLP* version of BT is poor because of severe cache thrashing, as shown in figure 3. The outcome is a slowdown of 1.32 over the single-threaded execution. In the case of FT, multithreading is beneficial for cache performance, however it yields

a marginal speedup of 1.03. The extensive resource sharing in Intel Hyperthreaded processors clearly does not allow effective exploitation of loop-level parallelism. For both benchmarks, the latency overlap achieved with multithreaded execution and the additional instruction-level parallelism do not measure up to the memory latency reduction achieved by precomputation. The overall performance of *SPR* is slightly better. *SPRINTS*, outperforms both *TLP* and *SPR*. Beyond the higher impact of *SPRINTS* on cache performance, the generation of efficient source code for precomputation results to smaller instruction streams and instruction working sets for the precomputation thread. This reduces the pressure on shared execution units, to the benefit of the computation thread. It must be noted that the magnitude of these speedups should be placed in the context of the capabilities of Intel's Hyperthreaded processors. The speedups attained with *SPRINTS* are comparable or higher than the speedups reported so far from physical experimentation with these processors [8].

4 Related Work

Research on SPR can be broadly classified into two classes: hardware-based SPR and software-based SPR. Hardware schemes identify accesses to precompute dynamically, by recording loads and their latencies at either the instruction fetch or the instruction retirement stage. Hardware schemes compose SPR code from the recorded delinquent loads and issue this code dynamically to hardware-triggered threads [3, 13]. The most aggressive hardware designs provide also a register communication mechanism to trigger SPR threads efficiently [12] without involving the operating system, and use manual or semi-automated construction of SPR instruction sequences. *SPRINTS* shares similarities with p-slices of Roth and Sohi [12] in that conceptually, both techniques try to derive highly optimized sequences of precomputation instructions and they both use results from simulation to drive the hadware/software precomputation engine. However, *SPRINTS* is a software technique which requires no hardware or compiler support.

Software SPR schemes can be based on programmer hints [10], compiler techniques [8] or binary modification techniques at load time [9]. Compiler and programmer-assisted techniques are more portable than binary modification techniques. Compiler techniques are preferable to programmer-assisted techniques because they are easy to use. *SPRINTS* shares this advantage with compiler techniques, but at the same time it differs in some important aspects: *SPRINTS* does not apply program analysis or runtime code profiling to detect delinquent loads, or perform any other SPR-specific optimization. It uses off-line cache simulation to identify all memory accesses that incur L2 misses and a compression grammar coupled with simple heuristics to pick those accesses that are responsible for dominant streams of L2 misses. The speculative streaming code is generated in the same high-level language as the sequential code, and can be optimized and executed efficiently from an unmodified compiler back-end and a standard multithreading runtime system. *SPRINTS* does not require program slicing, array access analysis, or other advanced compiler support to identify potential cache misses. Finally, *SPRINTS* targets specifically memory-bound scientific applications, which have not been targeted earlier compiler-based SPR schemes.

SPRINTS borrows the algorithm and the Sequitur grammar for compressing streams of delinquent memory references from earlier work on dynamic hot data stream pre-

fetching [2]. *SPRINTS* differentiates from dynamic hot data stream prefetching in the following aspects: First, *SPRINTS* uses offline analysis of traces of memory references that miss in the L2 cache, rather than online analysis of complete traces of memory references as they appear in the program. In other words, *SPRINTS* compresses traces of misses rather than traces of accesses. This decision is mandated by the tight time constraints of prefetching, which in turn calls for high prefetching accuracy and timeliness. Second, *SPRINTS* uses offline, rather than online analysis of traces. This is dictated by the use of simulation, which is an inherently slow technique for detecting streams of misses, but detects accurately such streams. An online application of *SPRINTS* would be possible with additional hardware support for buffering streams of cache misses and the associated target memory addresses. Intel Itanium processors provide such functionality [5]. Third, *SPRINTS* exploits simultaneous multithreading, while dynamic hot data stream prefetching uses a single-threaded prefetching mechanism. Finally, in contrast to dynamic hot data stream prefetching which targets sequential codes dominated by pointer-chasing, *SPRINTS* targets memory-intensive scientific codes, which are dominated by streams of memory references with predictable patterns.

5 Conclusions

This paper presented *SPRINTS*, a source-level streaming precomputation technique designed to improve the performance of memory-bound scientific applications on SMT processors with limited resources. Resource sharing often renders the execution engine incapable of achieving high-performance from regular, thread-level parallelization on these processors. *SPRINTS* requires no compiler or hardware support. It uses a compact representation of traces of cache misses and exploits this representation to associate delinquent memory accesses to data elements in the source code and produce highly efficient, source-level precomputation code. Experiments with realistic scientific applications show that *SPRINTS* clearly outperforms both TLP and indiscriminate speculative precomputation on Intel's Hyperthreaded processors. In the near future we plan to address a number of design and implementation issues of *SPRINTS*, including the use of lossy compression to improve the quality of streams by filtering out noisy irregular references, the use of mechanisms that can project the miss streams for multiple data inputs from one cache simulation with a single representative input, and the deployment of *SPRINTS* in multi-SMT systems.

Acknowledgements

This work is supported by an NSF CAREER Award (NSF CCF–0346867), an NSF ITR grant (NSF ACI–0312980) and the College of William and Mary.

References

1. T. Chilimbi. Efficient Representations and Abstractions for Quantifying and Exploitiing Data Reference Locality. In *Proc. of the 2001 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 191–202, Snowbird, UT, June 2001.

2. T. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General Purpose Programs. In *Proc. of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'2002)*, pages 199–209, Berlin, Germany, June 2002.
3. J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proc. of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, pages 306–317, Austin, TX, December 2001.
4. J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA–28)*, pages 14–25, Göteborg, Sweden, July 2001.
5. S. Eranian. The Perfmon2 Interface Specification. Technical Report HPL-2004-200R1, HP Labs, February 2005.
6. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
7. Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March/April 2004.
8. D. Kim and D. Yeung. A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code. *ACM Transactions on Computer Systems*, 22(2):326–379, 2004.
9. S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Bass Binary Adaptation for Software-Based Speculative Precomputation. In *Proc. of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'2002)*, Berlin, Germany, June 2002.
10. C. Luk. Tolerating Memory Latency through Software Controlled Preexecution on Simultaneous Multithreading Processors. In *Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 40–51, Göteborg, Sweden, July 2001.
11. Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), February 2002.
12. A. Roth and G. Sohi. A Quantitative Framework for Quantitative Pre-Execution Thread Selection. In *Proc. of the 35th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO–35)*, Istanbul, Turkey, November 2002.
13. K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 191–202, Cambridge, MA, November 2000.
14. UltraSPARC©IV Processor Architecture Overview. Technical report, Sun Microsystems, February 2004.
15. T. Wang, F. Blagojevic, and D. Nikolopoulos. Runtime Support for Integrating Precomputation and Thread–Level Parallelism on Simultaneous Multithreaded Processors. In *Proc. of the 7th ACM SIGPLAN Workshop on Languages, Compilers and Runtime Support for Scalable Systems (LCR'2004)*, Houston, TX, October 2004.