Feasibility of Dynamic Binary Parallelization

Jing Yang, Kevin Skadron, Mary Lou Soffa, and Kamin Whitehouse
Department of Computer Science
University of Virginia
{*jy8y, skadron, soffa, whitehouse*}*@cs.virginia.edu*

Abstract

This paper proposes DBP, an automatic technique that transparently parallelizes a sequential binary executable while it is running. A prototype implementation in simulation was able to increase sequential execution speeds by up to $1.96x$, averaged over three benchmarks suites.

1 Introduction

Fundamental issues in microprocessor technologies have led designers to increase the number of cores on a chip instead of increasing its single-threaded performance. Multi-core designs with 4 to 8 cores are ubiquitous, and trends suggest that core counts will continue to grow for the foreseeable future [24, 27]. Unfortunately, most existing software is designed for single-core processors, and is therefore unable to fully exploit the increased processing power offered by many-core processors. Thus, emerging microprocessor architectures are leaving behind a large and important base of existing software that represents years and sometimes decades of investment.

Existing parallelization technologies are not always practical for existing software. Many existing techniques require source code to be rewritten using parallel languages [14, 25] or libraries [26, 68], but this is often impractical due to cost: efforts to analyze, fix, and test existing software due to the Y2K bug alone are estimated to have cost about $20 billion in the 1990's [60], and rewriting code to find opportunities for parallelism would be a much larger task. Alternatively, *automatic parallelization* techniques do not require code to be rewritten, but they typically do require access to the source code for analysis. In many cases, all or some of the source code and development tool chain has been lost or, in the case of third-party software, was never available. Furthermore, software systems often involve components written in different programming languages, which makes cross-module parallelization difficult, if at all possible. Some parallelization techniques do not require source code and analyze the binary executable directly [19, 36, 73], but even these techniques are typically static and so cannot parallelize across dynamically linked executables and libraries, which are not known until run time and can change or be upgraded.

We are currently exploring a novel technique called *dynamic binary parallelization (DBP)* that can automatically parallelize software without access to the source code. DBP is based on the insight that many programs tend to frequently repeat long sequences of instructions called *hot traces* [47, 53]. DBP monitors a program at run time and dynamically identifies these hot traces, parallelizes them, and caches them for later use so that the program can execute in parallel every time a hot trace repeats. The main advantage of DBP is that it can transparently parallelize any sequential binary instruction stream; if the software can execute, then DBP can parallelize it. Thus, DBP can operate on any program, even when existing techniques have difficulty. For example, DBP can parallelize both legacy and third-party software when the source code is not available; it can parallelize across different program components that were originally written in different languages; it can also parallelize across dynamically linked executables and libraries. In this paper, we describe a prototype implementation in simulation and present preliminary evaluation results that demonstrate that the approach is promising.

2 Alternative Approaches

Several other approaches can transparently parallelize sequential binary executables, but each has important limitations. One natural approach is a distributed superscalar design. For example, core fusion [31] and core federation [65] use non-centralized hardware to combine simple cores to provide a wider superscalar width. However, scalability is limited by branch prediction, memory addresses, and instruction window size. In comparison,

the trace-based approach we are exploring has two advantages. First, holistically predicting a trace is easier than predicting multiple branches separately [32, 62], enlarging the effective instruction window. Second, a trace can be further optimized off-line to avoid pipeline stalls by rearranging the order of its instructions [28].

Another approach is to build a CFG by directly analyzing the binary [19, 29, 36, 70, 73]. This approach can exploit more coarse-grained parallelism, but to do so must take every possible execution path into consideration, also known as the *path explosion problem*. This can introduce many spurious data dependencies that reduce parallelism but do not appear in the execution path that is actually taken. In comparison, a trace-based approach can parallelize more aggressively because it must only consider a single execution path for each trace.

Previous research has used traces to produce new techniques for extracting more parallelism, including trace processing [59], dynamic multi-threading [3], and speculative multi-threading [45, 46]. More recently, similar approaches have been adopted by several Java virtual machines to extract parallelism from DOALL loops and recursive functions [9, 10, 11]. However, these approaches all speculate multiple consecutive tasks or traces, and run them in parallel. In our work, instead of predicting multiple consecutive traces, we use and improve techniques to find very long hot traces [53] and then parallelize within the trace instead of across traces. Not only does this simplify and improve prediction, it also enables off-line optimization of a much longer sequence of instructions.

3 Approach Overview

A conceptual overview of DBP is illustrated in Figure 1. Core 1 is equipped with trace management functionality and starts to execute the unmodified, sequential binary. Simultaneously, the *trace creator* monitors the instruction stream and identifies traces from frequently-repeating instruction sequences. The traces are then processed by the *trace parallelizer* and stored in the *trace cache*. This parallelization process is offloaded to spare cores or special accelerators in order not to affect the sequential execution. The optimization and parallelization algorithms ignore control dependencies introduced by *side exits* in order to fully exploit the parallelism in a trace. This allows for very aggressive parallelization, but also entails that traces must be run speculatively becase a parallelized trace can produce incorrect program state if a side exit is taken.

Therefore, at every point during execution, the *trace predictor* checks for *candidate traces*: parallelized traces in the trace cache that (i) begin with the instruction that is about to be executed by the sequential binary, and

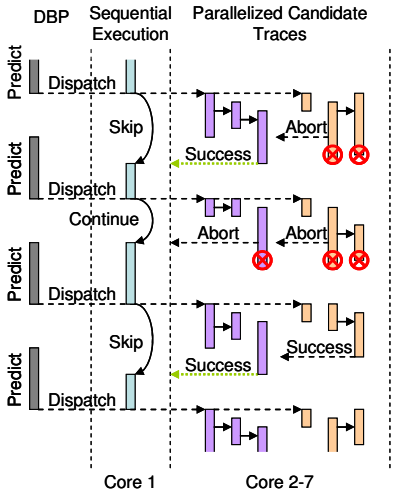

Figure 1: The DBP system uses one core for trace management plus sequential execution, and the remaining cores for speculative execution of parallelized candidate traces.

(ii) have a high probability to run to completion. If any exist, it suspends the sequential execution and launches them in the remaining available cores (Cores 2-7). The speculated traces operate on *copies* of the actual program state. If a trace reaches a side exit, it aborts and its copy of program state is discarded. If any traces run to completion, one of them is selected and its copy of program state is committed to the suspended sequential execution, which "skips forward" in time to the end of the selected trace. This *copy or discard* [67] mechanism eliminates the need for roll-back on failed speculation, which can be expensive for buffering the outputs of all speculative instructions. The figure illustrates three example scenarios. First, the right trace aborts and the left trace succeeds, causing the sequential execution to skip forward. Second, both traces abort and so the sequential binary continues running from the last dispatch point. Third, both traces succeed and the copy of program state from the left trace is chosen to commit.

DBP is a sub-class of *dynamic binary translation (DBT)*, which is a general approach to modify a binary during execution. Previously, DBT systems have demonstrated their benefits for compatibility [20, 22], profiling [43, 49, 72], security [30, 34], and performance [7, 12, 23, 48, 74]. However, unlike most DBT applications, DBP must create very long traces in order to be effective; short traces would not contain enough parallelism to outweigh the overhead of dispatching parallel tasks to different cores. Longer traces are more likely to have both fine-grained and coarse-grained parallelism. Another challenge for DBP is that speculative trace execution will certainly use more hardware resources and hence more energy in the processor than conventional ex-

ecution. However, even if DBP does require much more energy, single-threaded performance remains important for a wide variety of real-time and interactive applications. Furthermore, our results show that, even with extremely simple prediction techniques, the overall number of instructions executed by DBP is only $2.18x$ that of sequential execution.

4 Proof-of-Concept Prototype

To prove the feasibility of DBP, we created a proof-of-concept prototype by extending the SESC simulation framework [57]. The assumed architecture comprises a number of clusters and each cluster comprises 8 in-order cores. Clusters are connected via a mesh-based multi-hop network [66] and cores within each cluster are connected via a crossbar-based synchronization array [56]. Some of the cores are equipped with DBP functionality, such as Core 1 illustrated in Figure 1.

Trace creation occurs at the retire stage of each instruction, and has no additional overhead since it is not on the critical path of pipeline execution [52, 59]. We rely on internal code structures (e.g., loops, functions) to restrict the starting and ending points of each *primitive trace*, which may contain (i) one complete subroutine invocation, (ii) one or several iterations of the same loop, or (iii) one basic block of 16 or more instructions. Several short primitive traces are also chained together to form a *compound trace* until the minimum trace length of 64 instructions is reached. We start to exploit code structures in the outermost scope of the binary to maximize trace length. However, for each code structure, if the number of created traces exceeds that can be predicted simultaneously, it is abandoned and the next level of scope is entered. To give an example, we assume a simple program comprising a nested loop with the outer loop L_1 and two separate inner loops L_2 and L_3. Initlally, traces can only start with the head of L_1. When too many traces are created, the head of L_1 is no longer allowed to start traces and the heads of L_2 and L_3 are used instead, since they contain a smaller number of potential execution paths. The hierarchy of code structures is built by DBP when the program is initially loaded and filled on demand to an on-chip table with 16K entries.

The trace parallelization process comprises four steps. First, we build the SSA form to eliminate anti and output dependencies. Second, we aggressively apply *dynamic binary optimization (DBO)*, including constant propagation, value propagation, common subexpression elimination, redundancy elimination, and dead code elimination. These optimizations either eliminate unnecessary data dependencies or increase data dependency lengths, providing more parallelization opportunities. Third, we use the *modified critical-path* algorithm [71] to schedule instructions across many cores and insert necessary synchronizations to maintain the correct register and memory access order. Two memory references are only considered non-aliased if they access either different memory regions or their effective addresses have the same base register and different offsets. Finally, we perform graph-based register allocation [13] to each parallel task separately. All parallelized traces are stored in the main memory and only the metadata is stored on chip for quick accesses. In this implementation, we mainly extract ILP from the entire trace, which is essential to the assumed architecture in which every single core is in-order. In future work, we will explore how more coarse-grained parallelism can be extracted from traces.

Trace prediction occurs simultaneously with normal instruction fetch, and has no additional overhead [52, 59]. Inspired by multi-path execution [2, 6, 35], we simply launch all enabled traces that start with the next executed instruction, and use the multi-hop network to transfer live-in and live-out registers in bulk. If multiple traces run to completion, program state from the longest one is chosen to commit. Also, if a particular trace is mispredicted too many times in a row, it is disabled temporarily until being identified again. The L_1 data cache is used to hold the speculative program state and the trace execution is considered unsuccessful if a cache line replacement is encountered.

5 Evaluation

We evaluated our prototype using the SPEC2000 (both integer and floating point) and MediaBench benchmark suites. In our simulation, we made the following simplified architectural assumptions: 1) each instruction takes exactly 1 clock cycle to execute; 2) the synchronization array takes 1 clock cycle to access and has 8 request ports shared by all cores in the cluster; 3) the multi-hop network has 2-clock-cycles-per-hop latencies and can route up to 32 bytes at a time; 4) the overhead to actually parallelize each trace is not modeled since it can eventually be amortized if the program is executed long enough. We calculated the speedup of DBP over sequential execution using instruction count, and used arithmetic averages through the entire evaluation.

Figure 2 shows the speedup of DBP over sequential execution of three different configurations: (i) DBO with no parallelization to isolate its own benefits, (ii) DBO and 2-way parallelization using 64 cores (8 clusters), and (iii) DBO with 8-way parallelization using 256 cores (32 clusters). Thus, in all configurations, at most 32 candidate traces can be speculated simultaneously.

Although the measurements are based on several simplifying assumptions, the results are promising for DBP. The speedup of floating point benchmarks using 2-way

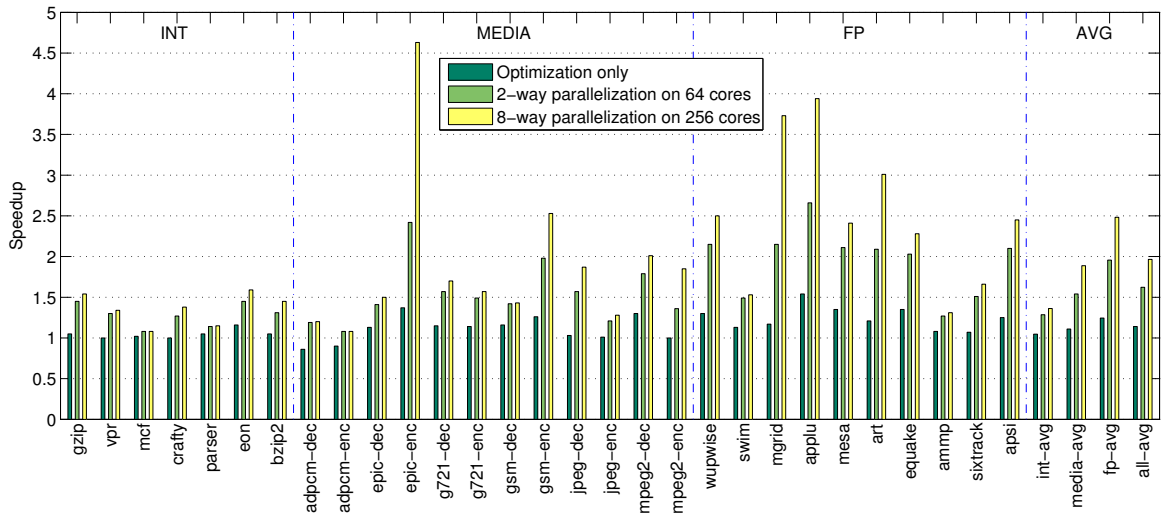

Figure 2: This figure shows the speedups of DBP over sequential execution by applying optimization only, 2-way parallelization on 64 cores, and 8-way parallelization on 256 cores, respectively.

parallelization can even exceed $2x$ for 7 out of 10 applications, because the optimization and parallelization gains are multiplicative. A speedup of at least $1.27x$ can be expected for 5 out of 7 integer benchmarks. This more modest speedup is not surprising, since integer programs normally have more complicated control flows that are hard to predict and pointer-based memory accesses that are hard to disambiguate. When using 8-way parallelization, DBP only achieves a propotional speed increase in a handful of benchmarks, such as *epic-enc*, $mgird$ and $applu$. This is mainly because spurious memory dependencies could not be effectively detected, and indicates an area for future exploration. The average speedup for DBO alone is $1.14x$.

One clear takeaway from the figure is that DBP performs best on floating point benchmarks, followed by media benchmarks, and could only achieve moderate speedup on integer benchmarks. Table 1 helps explain this difference with a statistical analysis of the trace creation and prediction algorithms. Column 3 shows the percentage of instructions executed by the unmodified program that are contained in correctly predicted traces. This number is typically over 90% for floating point and media benchmarks, which indicates that almost all dynamic instructions can be executed in a parallelized trace. However, the average percentage for integer benchmarks is only 61.35%. Column 4 shows the number of traces that are created during the entire program execution, which affects the cache size required. Although DBP creates many traces for a couple of benchmarks (e.g., $crafty$, $parser$), it also deletes most of them afterward. Thus, only a small number of traces actually reside in the trace cache when a program enters the steady state of execution. Also, the small gap between created and committed traces indicates that the current algorithm is sophisticated enough to only generate useful traces. Column 5 shows that the fraction of dispatches for which all traces aborted is typically less than 5-8$\%$. This indicates fairly high speculation accuracy, even with a very simple trace prediction algorithm. Furthermore, Column 6 illustrates that only 7 or fewer traces are actually dispatched on average, despite the possibility to execute up to 32 traces on our hardware models.

In practice, far fewer than 7 candidate traces actually run at any given time, because failed speculations end more quickly on average while successful speculations run to completion. Our results show that the overall number of instructions executed by DBP is only $2.18x$ that of sequential execution, averaged over all benchmarks, and in the worst case, it can be as high as $6.82x$ ($adpcm$-dec). Thus, speculative trace execution increases core utilization by a little more than the average performance improvement of $1.96x$, indicating a reasonable energy overhead. On the other hand, reducing the number of *possible* traces below 32 does affect performance, which indicates that sometimes dispatch of 32 traces is necessary. This opens an opportunity to combine DBP with multiprogramming to increase core utilization by multiplexing cores across multiple applications, dynamically allocating as many cores as an application can utilize.

Column 7 indicates that the average trace length varies drastically for each benchmark, ranging from 100s of instructions for integer benchmarks to 1000s of instructions for floating point benchmarks. The trace length is determined by the repeatability of paths in each program and, not surprisingly, is highly correlated to the speedup for each class of benchmarks shown in Figure 2. In future work, we will use insights from these statistics and

Benchmark		Exec. on Traces %	Trace #	Misp. Rate	Ave. Candidate #	Ave. Trace Length
INT	gzip	86.97 %	1,030 - 909 - 541	1.95 %	2.56	108.63
	vpr	80.90 %	1,176 - 1,085 - 627	3.92 %	4.40	125.33
	mcf	31.28 %	531 - 468 - 364	11.77 %	5.88	93.70
	crafty	61.48 %	13,008 - 12,569 - 11,122	2.53 %	2.93	78.58
	parser	32.99 %	7,228 - 6,844 - 6,100	4.30 %	3.12	101.17
	eon	74.53 %	575 - 544 - 247	3.03 %	3.51	122.10
	bzip2	61.31 %	958 - 798 - 476	7.61 %	3.84	112.86
Media	adpcm-dec	95.69 %	16 - 13 - 1	1.53 %	6.43	73.40
	adpcm-enc	97.10 %	28 - 26 - 5	0.42 %	5.82	83.06
	epic-dec	89.08 %	135 - 102 - 38	7.63 %	2.42	136.15
	epic-enc	96.52 %	131 - 117 - 42	8.11 %	2.75	862.21
	g721-dec	86.64 %	160 - 147 - 27	5.37 %	6.85	103.30
	g721-enc	70.55 %	197 - 185 - 108	5.27 %	6.68	105.52
	gsm-dec	97.93 %	80 - 74 - 7	8.06 %	4.23	1,098.80
	gsm-enc	97.23 %	134 - 131 - 6	2.54 %	3.21	756.13
	jpeg-dec	87.97 %	525 - 453 - 340	4.21 %	3.73	240.80
	jpeg-enc	59.23 %	762 - 708 - 530	2.55 %	2.78	138.48
	mpeg2-dec	91.31 %	608 - 567 - 326	2.17 %	2.70	175.21
	mpeg2-enc	68.81 %	678 - 540 - 319	10.56 %	4.61	394.59
FP	wupwise	99.12 %	82 - 78 - 6	2.24 %	2.40	2,179.74
	swim	96.74 %	68 - 52 - 16	4.38 %	2.52	836.16
	mgrid	99.85 %	207 - 194 - 10	0.19 %	1.13	7,890.64
	applu	97.58 %	100 - 85 - 7	0.96 %	1.84	4,583.59
	mesa	98.06 %	113 - 105 - 12	0.52 %	1.92	567.40
	art	99.07 %	67 - 64 - 3	0.76 %	1.96	3,986.86
	equake	95.55 %	231 - 221 - 94	2.60 %	4.25	638.74
	ammp	79.64 %	782 - 652 - 394	1.71 %	2.76	182.15
	sixtrack	89.88 %	1,809 - 1,656 - 897	0.70 %	1.85	119.37
	apsi	98.62 %	470 - 442 - 155	1.23 %	1.62	3,362.69

Table 1: This table shows 1) the percentage of instructions executed by the unmodified program that are covered by correctly predicted traces, 2) the total number of created traces, the number of traces that commit at least once, and the number of deleted traces, 3) the trace misprediction rate, 4) the average number of candidate traces for each prediction, and 5) the average length of the trace that commits in each correct prediction.

analyses to drive the development of new techniques for each individual class of programs.

6 Related Work

Traces have long been used to improve program performance. For example, a trace cache can increase the instruction fetch width [58]; a trace processor [59] speeds up control prediction by speculating on traces instead of branches; dynamic optimization [7, 12, 23, 48, 74] exploits optimization opportunities on traces which were not available on CFGs; clustered architectures rely on statically generated traces to schedule instructions among different functional units [16, 17, 42].

Hyperblocks are different from traces in that they contain multiple execution paths instead of a single one [44]. Instructions from different execution paths are guarded by hardware-supported predicates to maintain correct control flows. Recently, hyperblocks have been used by the TRIPS processor [62] to exploit ILP.

Static parallelization approaches analyze the source code to extract all parallelism at compile time [1, 4, 5, 37, 38, 50, 51, 55, 61, 75]. Lacking run-time information, these techniques must perform a conservative dependency analysis that includes dependencies from all possible paths through execution, and cannot adapt to other dynamics during execution.

Ever since the multiscalar architecture [63], thread-level speculation techniques have been used to release spurious dependency constraints caused by conservative static analysis [8, 21, 33, 40, 67, 69]. Generally, a high speculation accuracy requires heavy programmer annotation or comprehensive profiling, both of which can be difficult in practice.

Most dynamic parallelization approaches insert control logic into the source code statically, and use it to select the best strategy at run time [39, 41]. Binary translation has been considered to support legacy software [64], but due to the complexity of decompilation, such techniques have only been used to reconstruct simple program structures such as DOALL loops [73].

Several papers have argued for a JVM-like layer to dynamically optimize and parallelize programs [15, 18, 54]. However, such techniques assume a dominant programming language just as Java.

7 Conclusion

This paper proposes DBP and uses a prototype implementation in simulation to prove its feasibility. Experimental results indicated DBP as a promising approach for transparent parallelization, and also suggest areas of future work, such as creating longer traces for integer benchmarks and improving trace prediction algorithms.

References

[1] AHN, J., DALLY, W., KHAILANY, B., KAPASI, U., AND DAS, A. Evaluating the Imagine Stream Architecture. In *Proceedings of the International Symposium on Computer Architecture* (June 2004).

[2] AHUJA, P., SKADRON, K., MARTONOSI, M., AND CLARK, D. Multipath Execution: Opportunities and Limits. In *Proceedings of the International Conference on Supercomputing* (July 1998).

[3] AKKARY, H., AND DRISCOLL, M. A Dynamic Multithreading Processor. In *Proceedings of the International Symposium on Microarchitecture* (December 1998).

[4] ALEEN, F., AND CLARK, N. Commutativity Analysis for Software Parallelization: letting Program Transformations See the Big Picture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2009).

[5] ALLEN, R., AND KENNEDY, K. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[6] ARAGÓN, J., GONZÁLEZ, J., GONZÁLEZ, A., AND SMITH, J. Dual Path Instruction Processing. In *Proceedings of the International Conference on Supercomputing* (June 2002).

[7] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2000).

[8] BHOWMIK, A., AND FRANKLIN, M. A General Compiler Framework for Speculative Multithreading. In *Proceedings of the Symposium on Parallel Algorithms and Architectures* (August 2002).

[9] BRADEL, B., AND ABDELRAHMAN, T. Automatic Trace-Based Parallelization of Java Programs. In *Proceedings of the International Conference on Parallel Processing* (September 2007).

[10] BRADEL, B., AND ABDELRAHMAN, T. The Potential of Trace-Level Parallelism in Java Programs. In *Proceedings of the International Symposium on Principles and Practice of Programming in Java* (September 2007).

[11] BRADEL, B., AND ABDELRAHMAN, T. The Use of Hardware Transactional Memory for the Trace-Based Parallelization of Recursive Java Programs. In *Proceedings of the International Conference on Principles and Practice of Programming in Java* (September 2009).

[12] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization* (March 2003).

[13] CHAITIN, G. Register Allocation and Spilling via Graph Coloring. *SIGPLAN Notices 39*, 4 (April 2004).

[14] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (August 2007).

[15] CHANG, M., SMITH, E., REITMAIER, R., BEBENITA, M., GAL, A., WIMMER, C., EICH, B., AND FRANZ, M. Tracing for Web 3.0: Trace Compilation for the Next Generation Web Applications. In *Proceedings of the International Conference on Virtual Execution Environments* (March 2009).

[16] CHU, M., FAN, K., AND MAHLKE, S. Region-Based Hierarchical Operation Partitioning for Multicluster Processors. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2003).

[17] CHU, M., RAVINDRAN, R., AND MAHLKE, S. Data Access Partitioning for Fine-Grain Parallelism on Multicore Architectures. In *Proceedings of the International Symposium on Microarchitecture* (December 2007).

[18] CLARK, N. Why Should I Rewrite My Software When Dynamic Compilation Can Be Good Enough? In *Proceedings of the Workshop on Software Tools for Multi-Core Systems* (April 2008).

[19] DASGUPTA, A. *Vizer: A Framework to Analyze and Vectorize Intel x86 Binaries*. Master of Science Thesis, Rice University, November 2002.

[20] DESOLI, G., MATEEV, N., DUESTERWALD, E., FARABOSCHI, P., AND FISHER, J. DELI: A New Run-Time Control Point. In *Proceedings of the International Symposium on Microarchitecture* (November 2002).

[21] DING, C., SHEN, X., KELSEY, K., TICE, C., HUANG, R., AND ZHANG, C. Software Behavior Oriented Parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2007).

[22] EBCIOGLU, K., AND ALTMAN, E. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *Proceedings of the International Symposium on Computer Architecture* (June 1997).

[23] FAHS, B., BOSE, S., CRUM, M., SLECHTA, B., SPADINI, F., TUNG, T., PATEL, S., AND LUMETTA, S. Performance Characterization of a Microarchitectural Framework for Dynamic Optimization. In *Proceedings of the International Symposium on Microarchitecture* (December 2001).

[24] GIBBONS, P. Theory: Asleep at the Switch to Many-Core. `http://www.umiacs.umd.edu/~vishkin/T&MC5-2009/PRESENTATIONS/Gibbons.ppt`.

[25] GUSTAFSSON, N. Axum Language Overview. `http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Language%20Spec.pdf`.

[26] HEINEMAN, G. May Column: Multi-Threaded Algorithm Implementations. `http://broadcast.oreilly.com/2009/06/may-column-multithreaded-algor.html`.

[27] HELD, J., BAUTISTA, J., AND KOEHL, S. From a Few Cores to Many: A Tera-Scale Computing Research Overview. `ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`.

[28] HENNESSY, J., PATTERSON, D., AND GOLDBERG, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[29] HERTZBERG, B., AND OLUKOTUN, K. DBT86: A Dynamic Binary Translation Research Framework for the CMP Era. In *Proceedings of the Workshop on Parallel Execution of Sequential Programs on Multi-Core Architectures* (June 2009).

[30] HU, W., HISER, J., WILLIAMS, D., FILIPI, A., DAVIDSON, J., EVANS, D., KNIGHT, J., NGUYEN-TUONG, A., , AND ROWANHILL, J. Secure and Practical Defense Against Code-Injection Attacks Using Software Dynamic Translation. In *Proceedings of the International Conference on Virtual Execution Environments* (June 2006).

[31] IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture* (June 2007).

[32] JACOBSON, Q., ROTENBERG, E., AND SMITH, J. Path-Based Next Trace Prediction. In *Proceedings of the International Symposium on Microarchitecture* (December 1997).

[33] JOHNSON, T., EIGENMANN, R., AND VIJAYKUMAR, T. N. Min-Cut Program Decomposition for Thread-Level Speculation. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2004).

[34] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure Execution via Program Shepherding. In *Proceedings of the USENIX Security Symposium* (August 2002).

[35] KLAUSER, A., AND GRUNWALD, D. Instruction Fetch Mechanisms for Multipath Execution Processors. In *Proceedings of the International Symposium on Microarchitecture* (December 1999).

[36] KOTHA, A., ANAND, K., SMITHSON, M., YELLAREDDY, G., AND BARUA, R. Automatic Parallelization in a Binary Rewriter. In *Proceedings of the International Symposium on Microarchitecture* (December 2010).

[37] LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. Space-Time Scheduling of Instruction-Level Parallelism on A Raw Machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1998).

[38] LEE, W., PUPPIN, D., SWENSON, S., AND AMARASINGHE, S. Convergent Scheduling. In *Proceedings of the International Symposium on Microarchitecture* (November 2002).

[39] LINDERMAN, M., BALFOUR, J., MENG, T., AND DALLY, W. Embracing Heterogeneity – Parallel Programming for Changing Hardware. In *Proceedings of the Workshop on Hot Topics in Parallelism* (March 2009).

[40] LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. POSH: A TLS Compiler that Exploits Program Structure. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming* (March 2006).

[41] LLANOS, D., ORDEN, D., AND PALOP, B. Just-In-Time Scheduling for Loop-Based Speculative Parallelization. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-Based Processing* (February 2008).

[42] LOWNEY, G., FREUDENBERGER, S., KARZES, T., LICHTENSTEIN, W. D., NIX, R., O'DONNELL, J., AND RUTTENBERG, J. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing 7*, 1-2 (May 1993).

[43] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., JANAPAREDDI, V., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2005).

[44] MAHLKE, S., LIN, D., CHEN, W., HANK, R., AND BRINGMANN, R. Effective Compiler Support for Predicated Execution using the Hyperblock. In *Proceedings of the International Symposium on Microarchitecture* (November 1992).

[45] MARCUELLO, P., AND GONZÁLEZ, A. Clustered Speculative Multithreaded Processors. In *Proceedings of the International Conference on Supercomputing* (June 1999).

[46] MARCUELLO, P., GONZÁLEZ, A., AND TUBELLA, J. Speculative Multithreaded Processors. In *Proceedings of the International Conference on Supercomputing* (June 1998).

[47] MARS, J., AND SOFFA, M. L. Mats: MultiCore Adaptive Trace Selection. In *Proceedings of the Workshop on Software Tools for MultiCore Sysems* (April 2008).

[48] MERTEN, M., TRICK, A., NYSTROM, E., BARNES, R., AND HWU, W.-M. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots. In *Proceedings of the International Symposium on Computer Architecture* (June 2000).

[49] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 2007).

[50] OTTONI, G., AND AUGUST, D. Global Multi-Threaded Instruction Scheduling. In *Proceedings of the International Symposium on Microarchitecture* (December 2007).

[51] OTTONI, G., RANGAN, R., STOLER, A., AND AUGUST, D. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the International Symposium on Microarchitecture* (November 2005).

[52] PATEL, S., AND LUMETTA, S. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers 50*, 6 (June 2001).

[53] PATEL, S., TUNG, T., BOSE, S., AND CRUM, M. Increasing the Size of Atomic Instruction Blocks using Control Flow Assertions. In *Proceedings of the International Symposium on Microarchitecture* (December 2000).

[54] PENRY, D. You Can't Parallelize Just Once: Managing Many-core Diversity. In *Proceedings of the Workshop on Manycore Computing* (June 2007).

[55] RAMAN, E., OTTONI, G., RAMAN, A., BRIDGES, M., AND AUGUST, D. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the International Symposium on Code Generation and Optimization* (April 2008).

[56] RANGAN, R., VACHHARAJANI, N., VACHHARAJANI, M., AND AUGUST, D. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (September 2004).

[57] RENAU, J., FRAGUELA, B., TUCK, J., LIU, W., PRVULOVIC, M., CEZE, L., SARANGI, S., SACK, P., STRAUSS, K., AND MONTESINOS, P. SESC Simulator. `http://sesc.sourceforge.net`.

[58] ROTENBERG, E., BENNETT, S., AND SMITH, J. Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the International Symposium on Microarchitecture* (December 1996).

[59] ROTENBERG, E., JACOBSON, Q., SAZEIDES, Y., AND SMITH, J. Trace Processors. In *Proceedings of the International Symposium on Microarchitecture* (December 1997).

[60] RUCCIUTI, N. Y2K Cost Estimate Cut by $2 Billion. `http://news.cnet.com/Y2K-cost-estimate-cut-by-2-billion/2100-1091_3-235131.html`.

[61] RYOO, S., UENG, S., RODRIGUES, C., KIDD, R., FRANK, M., AND HWU, W. Automatic Discovery of Coarse-Grained Parallelism in Media Applications. *Transactions on High Performance Embedded Architectures and Compilers 1*, 1 (January 2007).

[62] SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S., AND MOORE, C. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the International Symposium on Computer Architecture* (June 2003).

[63] SOHI, G., BREACH, S., AND VIJAYKUMAR, T. N. Multiscalar Processors. In *Proceedings of the International Symposium on Computer Architecture* (June 1995).

[64] SRIVASTAVA, A., AND EUSTACE, A. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the Conference on Programming Language Design and Implementation* (June 1994).

[65] TARJAN, D., BOYER, M., AND SKADRON, K. Federation: Repurposing Scalar Cores for Out-Of-Order Instruction Issue. In *Proceedings of the Design Automation Conference* (June 2008).

[66] TAYLOR, M., LEE, W., AMARASINGHE, S., AND AGARWAL, A. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture* (February 2003).

[67] TIAN, C., FENG, M., NAGARAJAN, V., AND GUPTA, R. Copy Or Discard Execution Model For Speculative Parallelization On Multicores. In *Proceedings of the International Symposium on Microarchitecture* (November 2008).

[68] TROLLTECH. Thread Support in Qt. `http://doc.trolltech.com/4.2/threads.html`.

[69] VACHHARAJANI, N., RANGAN, R., RAMAN, E., BRIDGES, M., OTTONI, G., AND AUGUST, D. Speculative Decoupled Software Pipelining. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (September 2007).

[70] WANG, C., WU, Y., BORIN, E., HU, S., LIU, W., SAGER, D., NGAI, T.-F., AND FANG, J. Dynamic Parallelization of Single-Threaded Binary Programs using Speculative Slicing. In *Proceedings of the International Conference on Supercomputing* (June 2009).

[71] WU, M.-Y., AND GAJSKI, D. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems 1*, 3 (July 1990).

[72] YANG, J., ZHOU, S., AND SOFFA, M. L. Dimension: An Instrumentation Tool for Virtual Execution Environments. In *Proceedings of the International Conference on Virtual Execution Environments* (June 2006).

[73] YARDIMCI, E., AND FRANZ, M. Dynamic Parallelization and Mapping of Binary Executables on Hierarchical Platforms. In *Proceedings of the Conference On Computing Frontiers* (May 2006).

[74] ZHANG, W., CALDER, B., AND TULLSEN, D. An Event-Driven Multithreaded Dynamic Optimization Framework. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (September 2005).

[75] ZHONG, H., LIEBERMAN, S., AND MAHLKE, S. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-Thread Applications. In *Proceedings of the International Symposium on High Performance Computer Architecture* (February 2007).