Task Superscalar: Using Processors as Functional Units

Yoav Etsion[1], Alex Ramirez[1,2], Rosa M. Badia[1], Eduard Ayguade[1,2], Jesus Labarta[1,2], Mateo Valero[1,2]

[1]*Barcelona Supercomputing Center (BSC–CNS)*
[2]*Universitat Politècnica de Catalunya (UPC)*

Abstract

The complexity of parallel programming greatly limits the effectiveness of chip-multiprocessors (CMPs). This paper presents the case for task superscalar pipelines, an abstraction of traditional out-of-order superscalar pipelines, that orchestrates an entire chip-multiprocessor in the same degree out-of-order pipelines manage functional units. Task superscalar leverages an emerging class of task-based dataflow programming models to relieve programmers from explicitly managing parallel resources. We posit that task superscalar overcome many of the limitations of instruction-level out-of-order pipelines, and provide a scalable interface for CMPs.

1 Introduction

It is widely believed that power and technological constraints only allow processor performance to scale by increasing on-chip parallelism. Consequently, CMPs gain more and more popularity. However, the complexity of traditional parallel programming models is currently impeding programmer from harvesting performance [2].

But a promising new class of *task-based dataflow programming models* such as StarSs [3], RapidMind [22], and Sequoia [12], combine dataflow principles with task-level parallelism (TLP). These models implicitly schedule work and data, thereby relieving the programmer of explicitly managing parallelism.

These models share conceptual similarities with out-of-order superscalar pipelines, such as dynamic data dependency analysis and dataflow scheduling. Effectively, by using tasks as a basic work unit, this flow embodies a *hybrid von-Neumann / Dataflow execution model* [17].

In this paper we present the case for *Task Superscalar Pipelines*, an abstraction of ILP out-of-order superscalar pipelines that employs processor cores as functional units, and manage a CMP as a large-scale, task-level, out-of-order processor. Task superscalar provides a common execution platform for task-based dataflow programming models, by capturing their conceptual similarities to out-of-order superscalar execution. Importantly, these include hiding much of the internal parallelism from the programmer, and facilitating a unified CMP management layer.

The paper is organized as follows: Section 2 discusses related work. Section 3 advocates managed parallelism through the use of task-based dataflow programming models. Section 4 then outlines task superscalar pipelines, and Section 5 discusses how using tasks can overcome ILP limitations. Finally, we conclude in Section 6.

2 Related Work

The performance scalability of hardware parallelism, alongside the notoriety of explicit parallel programming, pushed the task of uncovering parallelism down to the compiler and hardware, which operate at ILP level. The most common examples of ILP designs are dynamically-scheduled out-of-order processors, which maintain a window of pending instructions, and dynamically schedule them in a dataflow manner [27].

The amount of ILP uncovered by an out-of-order processor directly depends on the size of its instruction window — which in turn motivated studies attempting to scale the effective window size. For example, Kilo-Instruction Processors [7] scale the window through segmentation and checkpointing, in order to overcome memory latencies. Alternatively, thread-level speculation techniques such as Multiscalar [34] and Trace Processors [30] split a large window into small speculative threads. Nevertheless, common wisdom suggests further scaling of ILP is not feasible.

In parallel, explicit dataflow architectures have been viewed as an alternative to the von-Neumann execution model. The past few decades have therefore seen intermittent studies on instruction-level dataflow architectures [1, 9, 11, 25, 26, 40]. However, inherent synchronization issues prevented such designs from scaling [17]. Moreover, these designs failed to support memory semantics of imperative languages such as C/C++ and Fortran [8].

The topic was recently revisited by TRIPS [6, 32] and WaveScalar [37], which put emphasis on supporting managed parallelism and imperative languages. But like previous ILP dataflow architectures, these designs are still susceptible to memory and communications latencies.

The recent shift towards CMPs spurred research into task-level parallel programming models. Such models include task-based dataflow programming models such as StarSs [3, 28], RapidMind [22], Sequoia [12], application specific dataflow backends [36], as well as streaming models such as StreamIt [38]. In contrast to task-based dataflow models which implicitly manage parallelism, streaming programming models require splitting the code into functional kernels, and explicitly expose the dataflow across kernels. The kernels are then mapped to computing elements, allowing data to *stream* through them. Variants such as OpenCL also support dynamic scheduling, but still need programs to explicitly express kernel and data dependences through event graphs.

CMPs have also motivated research into explicit hardware support for tasks, such as Carbon [21] and ADM [31], which use hardware task queues to support fast task dispatch and stealing, and MLCA [18] that manages global dependencies using a universal register file.

Task superscalar, proposed in this paper, provides coarse-grain managed parallelism through a dynamic dataflow execution model. Task superscalar supports imperative programming on large-scale CMPs without any fundamental changes to the micro-architecture.

3 Task-Based Managed Parallelism

Processor performance has historically scaled by increasing hardware parallelism and processing frequencies. With the latter nullified by power and technological constraints, parallelism is now the only means of scaling performance.

But parallel programming is not trivial. Programmers therefore typically prefer sequential programming, and not be burdened with explicit parallelism. This HW/SW disparity is known as the *parallelism gap*, and explicit parallel programming is still considered an artisan's job.

As a result, high-performance uniprocessors *manage parallelism* by providing the programmer with a sequential interface, and relying on either compilers or hardware runtime systems to uncover parallelism and manage parallel resources. This has proven very effective in high-performance ILP architectures, namely out-of-order [27] and VLIW [13]. The parallelism gap therefore motivates efforts towards large-scale *managed parallelism* [16].

Inevitably, ILP alone can no longer serve as a performance catalyst, as technological constraints prevent scaling instruction windows. Moreover, most available ILP is already used to hide memory latencies rather than parallel execution. Performance can therefore only scale by increasing the number of on-chip processors. In turn, the coarse-grain parallelism implied by CMPs brings forth the full impact of the parallelism gap.

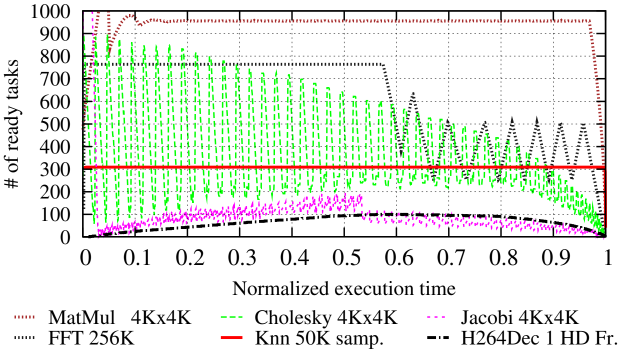

Figure 1: Available task parallelism in sample kernels, as uncovered by StarSs [3, 28] using a task window size of 8K tasks.

Task-level parallelism thus appears like a natural candidate for CMP-level managed parallel environments. Explicit annotations of function/task operands' directionality uncovers tasks' data requirements, thereby positioning tasks as abstract instructions with variable runtime.

This abstraction fosters adapting existing ILP dataflow analysis techniques to manage parallelism, either at statically, a-la VLIW, or dynamically, like out-of-order designs. This paper focuses on dynamic analysis.

3.1 Task-Based Dataflow Prog. Models

Task-based dataflow programming models enable programmers to tag functions in a manner that identifies their inputs and outputs to the runtime system. The runtime can then reason about data dependencies and execute the functions as tasks in a dataflow manner, thus relieving programmers from explicitly managing parallelism. Such models include StarSs [3, 28], RapidMind [22], Sequoia [12].

Tagging functions is performed by annotating their operands as either *input*, *output*, or both. As operands commonly consist of memory addresses (pointers), their directionality provides the function's dependency structure. Calling tagged functions requires no special syntax.

At runtime, a master thread begins executing the *main* function. Whenever an annotated function is invoked, the runtime system encapsulates the function code and the operand *values* with which it was called (where values can be either scalars or memory addresses) to create a deferred computational *task*. Values consisting of memory addresses are checked against previous uses of the memory regions they point to in order to identify inter-task data dependencies. The newly created task is then added to the task dependency graph. In addition, the runtime can also break anti-dependencies by renaming entire memory objects. Consequently, task execution is asynchronous, and spawning threads do not block and continue to execute (potentially spawning more tasks).

The runtime generated task graph enables the task scheduler to reason about data dependencies and plan task execution and data transfers ahead of time, thereby employing a dataflow execution model. For example, variants of *StarSs* [3, 4, 28] hide memory latencies by overlapping computation and data transfers.

Figure 1 demonstrate the effectiveness of these models in uncovering task-level parallelism, by showing the number of parallel tasks available throughout the execution of common computational kernels[1]. Parallelism was uncovered by analyzing a window of the most recently created 8K tasks (using StarSs/Cell B.E. task traces) — similar to the modus-operandi of out-of-order pipelines.

The figure shows that the models uncover dozens to hundreds of parallel tasks at most given times. This is most evident with kernels exhibiting simple task dependency structures such as MatMul, FFT, and Knn, all having hundreds of parallel tasks at all given times. But even kernels with complex dependency structures such as H264 (diagonal wavefront), Jacobi (neighboring block dependencies), and Choleski (multiple wavefronts) exhibit substantial task parallelism. For example, H264 peaks at $\sim$100 parallel tasks when its wavefront is at maximum width, whereas Choleski, despite its variability, enjoys over 200 parallel tasks during most of its execution.

In summary, these results demonstrate the premise of task-based managed parallelism, as an abstraction of instructions-level out-of-order pipelines. Moreover, the generality of such models makes them natural backends for higher-level paradigms, such as functional languages, *MapReduce* [10], Intel *Ct* [14], and *Cnc* [19].

Finally, it is important to note that these models effectively relax the memory model requirements, due to the availability of dataflow information. In turn, this gives way to the utilization of memory models such as DAG consistency [5] or bulk consistency [39]. DAG-like consistency is in fact used to manage non-coherent local memories in the Cell B.E. variant of StarSs [3].

3.2 Task Granularity

Task granularity presents a fundamental tradeoff between performance and programmability. While smaller tasks are typically more intuitive to the programmer, the overhead of decoding task dependencies can prevent these models from scaling. Indicatively, given a task decode latency of L, the minimal time required to decode enough tasks to utilize P processors is $T = L \times P$. Tasks whose runtime is shorter than T will therefore finish before a new task is assigned to the processor, thereby impeding processor utilization. Rico et al. measured L at $\sim$4us for the Cell B.E. variant of StarSs [29]. Repeating this measurement on a 2.66GHz Intel Core Duo averaged L at $\sim$1us. But even a 1us overhead imposes a minimum task runtime of 100us on a 100 processor CMP, suggesting that software-based runtime systems can only scale by greatly increasing task granularity.

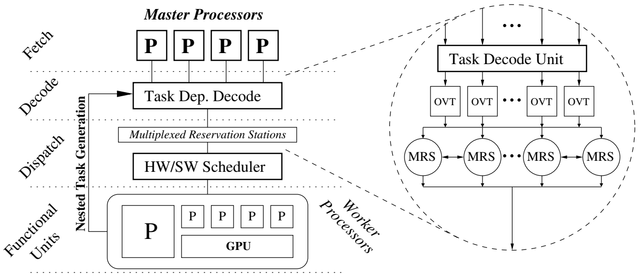

Figure 2: The proposed task pipeline, highlighting its frontend: the *Decode* and *Dispatch* stages, consisting of a *Task Decode Unit*, *Object Versioning Tables* (OVT), and the *Multiplexed Reservation Stations* (MRS).

On the other hand, architectural parameters also impose a minimal task runtime. For example, if the task runtime is shorter than the time required to transfer the task data and code to the designated processor, overlapping communication with computation becomes impossible and memory latencies prevail. On the Cell B.E. (256KB local stores connected to 25GB/s links [15]), this occurs when task runtime fall below 10us — an order of magnitude lower than the limit imposed by software-based runtimes.

The tradeoff between performance and programmability advocates research in two complementary tracks. These include designing hardware-based dependency decode mechanisms can reduce the overhead of decoding tasks dependencies by over an order of magnitude, thereby supporting fine-grain, easy to program tasks. In contrast, the performance scalability of coarse-grain tasks calls for research into optimization mechanisms that can adapt task granularity through task fusion and splitting, to accommodate the underlying architecture.

4 The Task Pipeline: Raising the Level of Abstraction

Task-based dataflow programming models aim to exploit parallelism at a granularity level much coarser than out-of-order ILP pipelines. Nevertheless, from an architectural viewpoint, tasks are no different than abstract instructions with variable runtime. In this section we highlight the similarities between the operational flow of such programming models and that of out-of-order pipelines, demonstrating the abstraction of out-of-order pipelines.

Figure 2 depicts the conceptual design of a task pipeline for heterogeneous CMPs, and maps it to a traditional out-of-order pipeline. Table 1 summarizes this mapping.

The task pipeline receives non-speculated tasks from either master or worker processors. Task pipelines need not rely on speculative execution, as operating at task granular-

[1] Kernels include blocked matrix multiplication (MatMul), fast Fourier transform (FFT), *Choleski* factorization, K-nearest neighbors classification (Knn), *Jacobi* solver, and decoding an H264 HD frame.

ILP Superscalar	Task Superscalar
Instruction (1ns–10ns)[1]	Task ($\geq$10us)[2]
Functional Unit (few)	Processors (hundreds)
Fetch	Task Generation
Decode	Task Dep. Decode
Reservation Stations (RS) (inst. in-flight: < 256)	Multiplexed RS (MRS) (tasks in-flight: $> 10K$)
Dispatch	Task Scheduler
Re-Order Buffer (ROB)	Out-of-Order Commit
Register file	L1 / Scratchpad Memory

[1] Estimated timing of scalar integer and floating-point operations.

[2] Based on the architectural parameters of the Cell B.E.

Table 1: Mapping task superscalar to ILP out-of-order pipelines.

ity neglects the overhead incurred by task generators when resolving branches leading up to the task generation.

The task decode stage then looks up the addresses of the task operands in object versioning tables listing the latest data producers. Tasks are then stored in multiplexed reservation stations (MRS) along with the explicit MRS mappings of its data producers, thus forming a distribted embedding of the explicit task graph. This flow is equivalent to that of out-of-order ILP pipelines, which includes decoding operand registers, looking them up in the register renaming table, and examining the register states. Tasks then typically wait in the MRS until all their data producers finish execution, much like instructions pending in the regular reservation stations.

A task scheduler determines the order in which tasks are executed, and on which processor. Supporting large task graphs promotes sophisticated task scheduling that can provides many runtime optimizations. For example, pending consumers can be sent to the processors executing their data producers, thereby saving precious bandwidth by eliminating data transfers between processor and memory [4]. Furthermore, we posit the scheduler can even target GPUs as processing backends, by co-scheduling instances of the same task type. In addition, the scheduler can be integrated on top of queuing systems that manage processor load through work stealing.

Finally, when task execution finishes, its output data written to memory, and its consumers updated. Notably, since tasks are *non-speculative*, they are retired out-of-order. This provides support for task windows considerably larger than out-of-order instruction windows, by eliminating the need for a reorder buffer.

The operational flow of task-based dataflow programming models is a straightforward abstraction of out-of-order superscalar pipelines. This analogy motivates research into implementing hardware *task pipelines*, thereby providing a CMP-wide layer of managed parallelism. Such exploration should draw from existing knowledge on out-of-order ILP pipelines. Furthermore, emphasis should be put on developing efficient (possibly software-based) scheduling algorithms, that can manage large-scale CMPs.

5 Overcoming ILP Limitations

The scalability of out-of-order pipelines is typically limited by the inability to efficiently implement and utilize large instruction windows [7, 24], which in turn limits the size of the dataflow graph they can observe, and amount of ILP they can uncover. These limitations can be attributed to conceptual issues arising from the execution model (branches, jumps, data anti-dependencies, exceptions), as well as issues arising from technological and timing constraints (synchronization, clocking, memory wall). The rest of this section discusses how changing work granularity can overcome the inherent limitations of ILP pipelines.

5.1 Task Pipeline Timing

A direct result of the change in work unit granularity is relaxing the timing constraints imposed on all the pipeline components. This is a result of fundamental queuing principles: the time needed by a functional unit to process an instruction dictates the rate in which instruction should be sent to the unit. The nanosecond granularity in which instructions operate, implies a synchronous implementation of all pipeline components using a global clock.

In contrast, task runtimes are measured in *microseconds*. Such granularities allow for asynchronous distributed designs, and do not require a global clock.

Relaxing the timing constraints therefore provide greater design flexibility, which in turn facilitates implementing task pipelines using scalable components.

5.2 Instruction Window Implementation

The design flexibility accommodated by the relaxed timing constraints calls for re-evaluating the non-scalable components of ILP pipelines — particularly the bypass network and single-entry reservation stations.

The strict timing constraints of ILP pipelines mandate the use of non-scalable *bus* topologies and broadcast-based communications, which greatly limits the number of reservation stations. In addition, latency constraints inhibit addressing inside reservation stations, thereby forcing a one-to-one mapping between in-flight instructions and reservation stations. As a result, typical ILP pipelines do not scale beyond 128–256 in-flight instructions.

Conversely, the relaxed timing constraints of task pipelines support replacing the bypass network with a scalable switched NoC, which can support hundreds of reservation stations and processing units.

In addition, relaxing access times to the reservation stations facilitates storing many tasks in a single station using dense eDRAM blocks. As task meta-data requires 128B–1KB of storage, implementing the stations using 256KB eDRAM blocks supports up to 2048 tasks per station.

Finally, task pipelines need not support precise exceptions [35, 33]. Such support, prevalent in ILP pipelines, requires a pipeline checkpointing mechanisms whose complexity increases with instruction window size. But the nature of exceptions relieves task pipelines from having to support them at a global level: *I/O Interrupts* can be handled by dedicated processors, whereas *Traps* and many *Exceptions* (page-faults) are task dependent and should be handled by the assigned processor. Only some *Exceptions* might have a global effect, but as these typically represent erroneous conditions (division by zero, for example), they can be handled by a software runtime.

5.3 Window Utilization and Speculation

The main runtime limitations of ILP pipelines is sustaining a high instruction throughput given the frequency of branch instructions. High performance processors therefore employ *Branch predictors* to speculate on the likely execution path [41]. Such pipelines typically predict the results of multiple outstanding branches (deep speculation). As a result, most in-flight instructions are speculated, and many are eventually discarded due to misspeculations. Hence, even if larger instruction windows could be implemented, they cannot be effectively utilized.

Task pipelines on the other hand, are *non-speculative*, and employ an asynchronous execution model. As opposed to ILP pipelines which use a *pull* model to fetch speculated instructions, task pipelines employ a *push* model in which the spawning thread/task explicitly pushes tasks into the pipeline. This distinction is more than semantical, as it transfers the burden of branch resolution to the task issuer. This also it provides straightforward support for multiple task generators and nested tasks.

Therefore, the non-speculative nature of task pipelines, and its support for multiple task generators, facilitates sufficient task throughput to support large-scale parallelism.

5.4 Overcoming The Memory Wall

ILP pipelines are already struggling with the memory wall, and effectively utilize ILP to hide increasing memory latencies rather than for parallel execution. In addition, all components in ILP pipelines share a common memory data link through the L1 cache. But as power and complexity limitations prevent increasing the number of cache ports, the memory bandwidth available to the pipeline is limited. Alternatively, some designs employ large register files to increase the *data* window available to functional units. But as register files have already become a power bottleneck [23], increasing their size is not feasible.

The memory wall is therefore in conflict with the von-Neumann execution model, as it breaks the model's random-access memory assumption. This understanding motivated studies into alternative execution models such as dataflow [1, 9, 11, 25, 26, 40] and intelligent RAM [20].

Task pipelines in contrast, provide a execution model in which individual tasks execute under the von-Neumann model, but are globally orchestrated in a dataflow manner. The random access assumption therefore need not be adhered globally, but only at the individual processor level. As global dataflow information enables processors to employ aggregate data transfers (either through DMAs to a local memory, or prefetching data to a local cache), tasks are presented with a random-access data memory, in accordance with the von-Neumann model. Furthermore, as data is transfered directly from the memory system to the processors, the task pipeline itself is not a bandwidth bottleneck for the entire system.

Finally, the relaxed timing constraints allow task pipelines to make use of software schedulers, as opposed to limited hardware schedulers used in ILP pipelines. This allows for more flexible scheduling algorithms that can be better tuned to data locality, memory bandwidth, heterogeneous processing, etc. For example, schedulers can save precious off-chip memory bandwidth by reordering tasks to facilitate direct data transfers between processors' local memories. The local memories/caches can therefore viewed as a distributed meta-register file, Another possible optimization enabled by the availability of dataflow information and aggregate data transfers, is hiding memory latencies by double buffering local memories and overlapping data transfers with the computation of a previous task.

In summary, task pipelines leverage the coarse granularity of tasks to hide memory latencies and distribute memory load across the CMP, and thereby avoid scalability bottlenecks to which ILP pipelines are so susceptible.

6 Conclusions & Open Research

We have presented the case for *Task Superscalar*, a task-level abstraction of out-of-order superscalar pipelines, that uses processors as functional units. Task superscalar builds on an emerging class of programming models that enable programmers to annotate task inputs and outputs. Much like ILP out-of-order pipelines, task superscalar dynamically analyzes task inputs and outputs to uncover data dependencies, identifies task parallelism, and schedules tasks in a dataflow manner. It is shown that this approach can uncover hundreds of parallel tasks in common computational kernels, and thereby has the potential to manage large-scale CMPs. Consequently, task superscalar provides a unified platform for both task and data parallelism.

The premise of task superscalar motivates research that will facilitate efficient hardware implementations. This includes hardware support for task dependency decoding, novel task and data scheduling algorithms, and task-level compiler optimizations. Such research should draw from existing knowledge on out-of-order ILP pipelines, aiming to provide task-level managed parallelism.

We are currently engaged in the design and implementation of a Task Superscalar pipeline.

Acknowledgments

This research is supported by the Consolider contract number TIN2007-60625 from the Ministry of Science and Innovation of Spain, the European Network of Excellence HIPEAC-2 (ICT-FP7-217068), the ENCORE project (ICT-FP7-248647), and the TERAFLUX project (ICT-FP7-249013). Y. Etsion is supported by a Juan de la Cierva Fellowship from the Spanish ministry of science.

References

[1] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. on Computers*, 39(3):300–318, Mar 1990.

[2] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. TR EECS-2006-183, UC Berkeley, Dec 2006.

[3] P. Bellens, J. Perez, R. Badia, and J. Labarta. CellSs: a programming model for the Cell BE architecture. *SC Conf.*, Nov. 2006.

[4] P. Bellens, J. M. Perez, F. Cabarcas, A. Ramirez, R. M. Badia, and J. Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Prog.*, 17(1-2):77–95, 2009.

[5] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. DAG-consistent distributed shared memory. In *Intl. Parallel Processing Symp.*, pp. 132–141, Apr 1996.

[6] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37:44–55, 2004.

[7] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-Instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, 2005.

[8] D. E. Culler, K. E. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In *Intl. Conf. on Parallel Arch. and Compilation Techniques*, pp. 153–164, 1993.

[9] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *Intl. Symp. on Comp. Arch.*, pp. 210–215, 1978.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symp. on Operating Systems Design & Impl.*, pp. 137–150, Dec 2004.

[11] J. B. Dennis and D. Misunas. A preliminary architecture for a basic data flow processor. In *Intl. Symp. on Comp. Arch.*, pp. 126–132, 1974.

[12] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. *SC Conf.*, 2006.

[13] J. A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *Intl. Symp. on Comp. Arch.*, pp. 140–150, Jun 1983.

[14] A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and C. B. Future-proof data parallel algorithms and software on intel multi-core architecture. *Intel Technology Journal*, 11(4), Nov 2007.

[15] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *Symp. on High-Performance Comp. Arch.*, pp. 258–262, 2005.

[16] W.-M. W. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, A. A. M. Sara S. Baghsorkhi, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicit parallel programming models for thousand-core microprocessors. In *Ann. Conf. on Design Automation*, June 2007.

[17] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Intl. Symp. on Comp. Arch.*, pp. 131–140, May 1988.

[18] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A multilevel computing architecture for embedded multimedia applications. *IEEE Micro*, 24(3):56–66, May-June 2004.

[19] K. Knobe. Ease of use with concurrent collections (CnC). In *Hot Topics in Parallelism*, Mar 2009.

[20] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9):75–78, 1997.

[21] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Intl. Symp. on Comp. Arch.*, pp. 162–173, 2007.

[22] M. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *Proc. GSPx Multicore Applications Conf.*, Oct 2006.

[23] F. J. Mesa-Martinez, M. Brown, J. Nayfach-Battilana, and J. Renau. Measuring performance, power, and temperature from real processors. In *Workshop Exp. Comp. Sci.*, p. 16, Jun 2007.

[24] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Intl. Symp. on Comp. Arch.*, pp. 206–218, Jun 1997.

[25] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Intl. Symp. on Comp. Arch.*, pp. 82–91, 1990.

[26] G. M. Papadopoulos and K. R. Traub. Multithreading: a revisionist view of dataflow architectures. In *Intl. Symp. on Comp. Arch.*, pp. 342–351, 1991.

[27] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, a new microarchitecture: rationale and introduction. In *Intl. Symp. on Microarch.*, pp. 103–108, 1985.

[28] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Intl. Conf. on Cluster Computing*, pp. 142–151, Sep 2008.

[29] A. Rico, A. Ramirez, and M. Valero. Available task-level parallelism on the Cell B.E. *Scientific Prog.*, 17(1–2):59–76, 2009.

[30] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. *Intl. Symp. on Microarch.*, p. 138, 1997.

[31] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Intl. Conf. on Arch. Support for Prog. Lang. & Operating Systems*, pp. 311–322, 2010.

[32] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Intl. Symp. on Microarch.*, pp. 480–491, 2006.

[33] J. Smith and A. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans. on Computers*, 37(5):562–573, May 1988.

[34] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Intl. Symp. on Comp. Arch.*, pp. 414–425, 1995.

[35] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Intl. Symp. on Comp. Arch.*, pp. 27–34, 1987.

[36] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC Conf.*, pp. 1–11, 2009.

[37] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Intl. Symp. on Microarch.*, p. 291, Dec 2003.

[38] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Intl. Conf. on Compiler Construction*, pp. 179–196, Apr 2002.

[39] J. Torrellas, L. Ceze, J. Tuck, C. Cascaval, P. Montesinos, W. Ahn, and M. Prvulovic. The Bulk Multicore architecture for improved programmability. *Comm. ACM*, 52(12):58–65, 2009.

[40] I. Watson and J. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, Feb 1982.

[41] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Intl. Symp. on Microarch.*, pp. 51–61, 1991.