The Road to Parallelism Leads Through Sequential Programming

Gagan Gupta Srinath Sridharan Gurindar S. Sohi
Department of Computer Sciences
University of Wisconsin-Madison
{*gagang,sridhara,sohi*}*@cs.wisc.edu*

Abstract

Multicore processors are already ubiquitous and are now the targets of hundreds of thousands of applications. Due to a variety of reasons parallel programming has not been widely adopted to program even current homogeneous, known-resource multicore processors. Future multicore processors will be heterogeneous, be increasingly less reliable, and operate in environments with dynamically changing operating conditions. With energy efficiency as a primary goal, they will present even more parallel execution challenges to the masses of unsophisticated programmers. Rather than attempt to make parallel programming more practical by chipping away at the multitude of its known drawbacks, we argue that sequential programs and their dynamic parallel execution is a better model. The paper outlines how to achieve: (i) dynamic parallel execution from a suitably-written statically sequential program, (ii) energy-efficient execution by dynamically and continuously controlling the parallelism, and (iii) a low-overhead precise-restartable parallel execution.

1 Introduction

The computing landscape continues to evolve at a phenomenal rate, placing new demands on hardware and software design. A decade ago, when multicore processors became commonplace, desktops (and traditional servers) accounted for the bulk of computing and performance was still the primary design objective (though energy efficiency was rapidly gaining importance). A relatively small number of software vendors typically created "shrink-wrapped" applications on desktops. Today, mobile devices have significantly expanded computing at the low end, and cloud computing at the high end. Multicore processors are being used across this spectrum. Energy efficiency has now become the primary design objective. The number of software vendors has increased dramatically, creating hundreds of thousands of applications to run on these diverse computing devices. Thus the importance of the ease of writing programs that will run in parallel (to achieve energy efficiency and/or performance) on a diverse set of devices continues to increase.

Most of the computing community has responded to the parallelism challenge by trying to increase the prevalence of parallel programming—to make parallel programming synonymous with programming. There has been a plethora of work in this direction, at all levels. Examples include introducing parallel programming into introductory classes so that, with the passage of time, parallel programming will become the default [5], providing parallel languages and libraries to alleviate the burden of parallel programming [3, 10, 11, 17, 21, 22, 33, 37], providing tools to address the complexities of parallel programs (e.g., non-determinism) [9,18,24,25,41,44] as well as models to facilitate the creation of parallel programs [21, 37].

While the research community endeavors to make parallel programming practical, the characteristics of computing environments (both hardware and software) as well as the likely applications of computing continue to change in meaningful ways that will make the goal daunting. Computing hardware is already transitioning from homogeneous processing cores, which are the default assumption for parallel processing, to heterogeneous processing cores with disparate energy consumption/performance characteristics [1]. Going further, pushing the limits of semiconductor technology will make computing hardware increasingly unreliable [2], and thus the capabilities of the available pool of resources will vary dynamically. Techniques to manage heat, temperature, and energy, as well as demands of other (statically unknown) software applications will further increase the uncertainty in the precise knowledge of resources available to execute a program. With more use of techniques like server consolidation, the mix of the different simultaneously running applications will in-

crease and to provide differentiated levels of service it is likely that resources (e.g., cores) available to an application (especially a lower-priority one) will change dynamically and often without notice [42]. In short, it will be unreasonable to expect the availability of a given (or constant) number of resources (e.g., cores) to process an application, a norm for contemporary parallel programming techniques, as well as operating systems.

Coupled with the uncertainty in available hardware resources will be an increasing number of programmers. No longer will most software be written by a small number of teams of experts. Rather, software will be created by a multitude of programmers who will rely on techniques, such as abstraction, encapsulation, and modularity that have been taught for decades. They will know about parallel algorithms and parallelism, but will not want to write statically-parallel programs. They will want their applications to run in a variety of different computing scenarios without detailed (or perhaps even any) knowledge of the multitude of different microarchitectures (or even ISAs), or their fault characteristics, or the software environments on which the dynamic instances of the application programs may run. Yet, to achieve performance and energy goals, parallel execution will have to be exploited on these systems.

The multi-objective goal then is to *write ubiquitous programs for which reliable, energy-efficient, parallel execution can be achieved while remaining agnostic of the dynamically (and potentially continuously) changing computing scenarios*. We don't believe this will be achieved by parallel programming techniques. Rather, we believe sequential programs and their controlled dynamic parallel execution presents a better alternative. A few recent proposals have made this case [8, 23, 35], but currently they do not take a comprehensive view of all of the above challenges. In this paper we present a model (§2) that seamlessly addresses all of the above aspects. It parallelizes execution of statically-sequential programs, and further controls the execution to accomplish performance and energy goals in dynamically changing (§3) and unreliable (§4) computing environments.

2 The Model

The model we propose aims to provide a practical paradigm to compose programs in the form of sequential programs, expressed in a familiar and established imperative programming language, such as C++. We propose to execute the programs employing dataflow execution, since it naturally exposes all of the innate parallelism within a program, but also control it to achieve our stated objectives. We elaborate further below.

2.1 Composing Programs

Programmers express parallel algorithms and explicitly orchestrate their parallel execution in traditional parallel programming. They typically employ the following three steps in the process: (i) identify appropriately-sized computations and the data shared between them, and compose the computations into tasks using algorithms and data structures amenable to concurrent execution, (ii) schedule the execution of the tasks within the program, and (iii) ensure that only independent tasks execute concurrently. The first two steps mostly impact the execution efficiency. The third step impacts correctness of the program and can prove to be significantly more challenging. Furthermore, most of the analysis and reasoning performed to design the programs (for efficiency as well as correctness) are based on statically determined worst-case scenarios, making it difficult to take advantage of dynamic parallelism opportunities and account for dynamic operating conditions of the system. Recent task-based models have come to provide assistance with the second step, such as support for design patterns [21, 32, 37] which can be complex to implement otherwise, easing some aspects of programming.

At the same time, programmers today follow modern software engineering and object oriented (OO) design principles [12]. For example, they leverage abstraction and compose programs from reusable modules (in the form of functions). Consequently functions act as self-contained computations that manipulate "hidden" data but communicate with each other using well-defined interfaces. Manipulating global data not communicated through the interface is often avoided and hence most such computations are free from side-effects.

The proposed model seeks to exploit the above programming practices followed by programmers. We observe that encapsulated functions are already well-suited for parallel execution. Their coarser granularity makes for a more suitable unit of computation for multicores. Hence the model exploits function-level parallelism. We realize not all imperative programs may follow OO principles, due to legacy or other reasons. Hence we provision to handle the rare "poorly" composed program, as described later. Next, of the three programming steps identified above, we rely on the programmers to provide for only the first step, leveraging their natural insights into their algorithms, while the model accomplishes the second and third steps. Thus we decouple the expression of a parallel algorithm from its execution, easing much of the burden on the programmers.

Programs written for the model closely resemble their sequential versions intended to run on a uniprocessor. For example Figure 1 shows the main loop of bzip2 [20] (coded in a current prototype of the model, implemented

as a run-time library), following the above design principles. Ignore the *df_execute* mnemonic on lines 8 and 10 (artifacts of the run-time library), for now; the function calls, *compress* and *write*, as they would appear in the sequential code, are shown in the comments on lines 7 and 9. The loop reads a block of data from an input file (line 5), compresses it (*compress*, line 7) and writes the results to an output file (*write*, line 9), repeating the process for all blocks. Although a program so composed is sequential, we seek to exploit its inherent parallelism, without using explicit threads or synchronization primitives, and hence programs in the model are *statically-sequential* (as opposed to *statically-parallel* in the multithreaded models). More significantly, the model ensures the execution proceeds as per the implied semantics that programmers have come to expect from sequential programs. How we do so is described next.

```
1  ..
2  op_set->insert (OpFile);    // File wr set
3  while (blockBegin < fileSize -1) {
4     blockBegin += updateBlock (blockBegin);
5     block = new block_t (InFile, Length);
6     block_set->insert (block);
7     // compress(block)
8     df_execute(block_set, &compress);
9     // write(OpFile, block)
10    df_execute(op_set, block_set, &write);
11 }
12 ..
```

Figure 1: Main loop of bzip2 coded in the model.

2.2 Executing Programs

The model treats a function as a unit of computation. It relies on the programmers to identify "dataflow" functions suitable for concurrent execution based on the knowledge of their algorithm, e.g., functions *compress* and *write* in Figure 1. To execute a program the model sequences through it, a function (rather than an instruction) at a time, and attempts the function's concurrent execution with other functions. Hence before executing the function we first need to ascertain its "operands". We also raise the granularity of an operand from an individual register or memory location to an object, which usually forms a basic unit of data in modern programs. A function's input operands (*read set*) and output operands (*write set*), also collectively called its *data set*, are readily available from its interface (e.g., the parameters to *compress* and *write* functions on lines 7 and 9 in Figure 1). Often objects in these sets may be unknown statically. Therefore, the model relies on the programmer to provide the computation that formulates the read and write sets and pass them to the function. For example, the object *OpFile* is included in the write set and the object *block* is included in the read set of the function *write* on line 9. At run-time, before the function is invoked, the model computes the read and write sets, by dereferencing pointers, allowing us to handle data referenced using pointers (e.g., the dynamic instances of the object *block* on line 5 in Figure 1) and pointer arithmetic.

We use the identity of the objects in the data set to establish the data dependence between functions, in contrast to independence as is established in the statically-parallel model. In particular, we determine if the function currently being processed is dependent on any prior function(s) that are still executing. If not, it is scheduled for execution. If so, its execution is suspended until its dependences are resolved. In either case, we continue to process the subsequent program.

A "poorly" composed function, e.g., one with unknown side effects or a third-party function, may make it difficult to ascertain its data set. In such a case the model can resort to its sequential execution, precluding the need to determine its precise data dependences, and hence its data set.

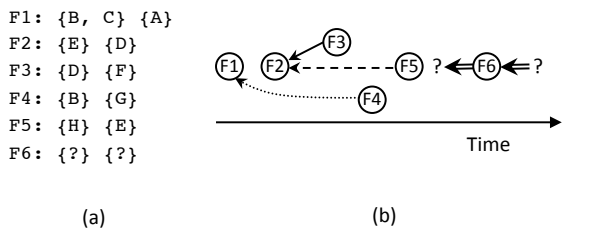

Figure 2: (a) Dynamic function invocations. *F:*{*wr_set*}{*rd_set*} modifies (reads) objects in its write (read) set. (b) Dataflow graph of the functions.

Consider a program composed of dynamic invocations of functions along with their dynamically computed write and read sets as shown in Figure 2a. Figure 2b shows the data dependence between the invocations, e.g., F3 writes objects D, and thus has a WAR dependence on F2 which reads D. Likewise F4 has a WAW dependence on F1, and F5 has a RAW dependence on F2. The data set of F6 cannot be ascertained and hence its dependences are unknown. As it processes the program the model discovers these dependences dynamically and honors them to maintain the sequential appearance of the static program. The model further strives to realize the optimum, i.e., the dataflow, schedule of execution in which independent functions F1 and F2 may execute concurrently, F3 may execute concurrently with F1, F4 and F5 but only after F2 has completed, F4 may execute concurrently with F2, F3 and F5 but only after F1 has completed, and F5 executes after F2 has completed, possibly in parallel with F1, F3 and F4. This is in contrast to the

statically-parallel model in which a programmer would have to conservatively account for the dependences, reasoning about all the possible execution schedules and interactions between the computations, an error-prone process that can stump even experts [27]. No longer is the onus on the programmer to orchestrate an efficient execution schedule.

The model begins program execution, similar to a sequential program, by sequencing through it on an available processor, executing operations and seeking dataflow functions. When such a function is encountered, it attempts to execute it concurrently with currently executing function(s), and the program continuation, i.e., remainder of the sequential program past the function, dependences permitting. The process is repeated with the program continuation on an available processor and thus the program execution unravels in parallel with its computations. As the program is unraveled, we control the amount of parallelism that is unwound, to realize a diverse range of objectives. It permits management of resources to prevent deadlocks, improve execution efficiency, and realize precise-restartable execution, of which the latter two we further discuss in §3 and §4.

Here we briefly describe our current runtime-library prototype of the model. To use the library programmers identify dataflow functions through a *df_execute* library call, e.g., on lines 8 and 10 in Figure 1. The prototype tracks the objects being accessed by the functions to manage data dependences. In particular, it tracks a function's start and completion of its access to an object. Programmers annotate the objects shared between functions when the objects are declared. They also group them into write and read sets, e.g., on lines 2 and 6, Figure 1, and pass them to the function as arguments via *df_execute* (lines 8 and 10).

The prototype employs a token protocol to identify and manage the data dependences. It associates tokens with objects and assigns each object as many read tokens as the machine's data width will permit and a single write token. Read tokens are used to manage consumption of the object while the write token is used to manage its production. When the execution encounters a dataflow function, it requests read (write) tokens for objects in the functions read (write) set; it is ready for execution only after it has acquired all its tokens. Upon completion, it relinquishes the tokens which are then passed to the suspended function(s), if need be. When a suspended function has acquired its requisite tokens, it can be scheduled for execution.

The prototype uses the work-first principle and lazy task creation to discover work only when a resource is idle to optimize utilization [7]. Work deques and randomized work-stealing are used to balance the load. As the program is sequenced, dependent functions are shelved, and they are introduced into the deques after their dependences have resolved [23].

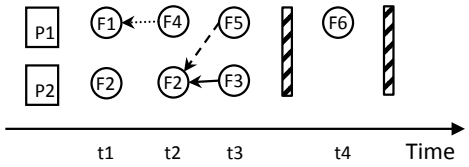

Figure 3: Example dataflow schedule of execution.

Figure 3 shows a possible execution schedule of the functions in Figure 2a on a system with two cores. The execution first encounters F1 and schedules it for execution. Next F2 is processed, and since it is independent, it too is scheduled for execution, concurrently with F1 (time *t1*, Figure 3). However, F3, encountered next, is suspended due to its dependence on F2, and likewise so are F4 and F5. After F1 completes (end of time *t1*) F4 is scheduled for execution (time *t2*) since its dependences have now resolved. Similarly other invocations are executed as shown in the figure. When F6 is encountered (identified by the programmer as one with an unknown data set) the model ensures that F6 executes only after all the preceding functions have completed. Execution past F6 proceeds only after F6 completes.

Thus by suspending dependent functions until their dependences have resolved, and executing other dynamic independent functions in the meantime we achieve dataflow execution from a statically-sequential program. The execution of such suitably-written programs is race-free, relieving the programmer from reasoning about aspects of execution such as the memory consistency model of the underlying hardware.

Execution of the bzip2 example in Figure 1 will unfold so that dynamic instances of the *compress* functions will execute concurrently, since they operate on disjoint data. Dataflow execution automatically sets up a 3-stage pipeline of computations in which a block is read from the file, compressed and written to the output file, without requiring the programmer to resort to special design patterns (e.g., pipeline parallelism [32]). Further, the dynamic invocations of the *write* function automatically serialize, since they write to the same object (the file handle), in the program order, achieving in-order file writes, precluding the need for any special handling.

Statically-sequential applications (blackscholes, barneshut, bzip2, dedup, histogram, and reverse index) from standard benchmark suites, developed using the model and run on three stock multicore machines, an 8-thread Intel Nehalem-based machine, a 16-core and a 32-core AMD Opteron-based machines, achieved speedups (harmonic mean) similar to their Pthread versions on the

Nehalem machine and over 20% better on the AMD Opteron machines (Figure 4).

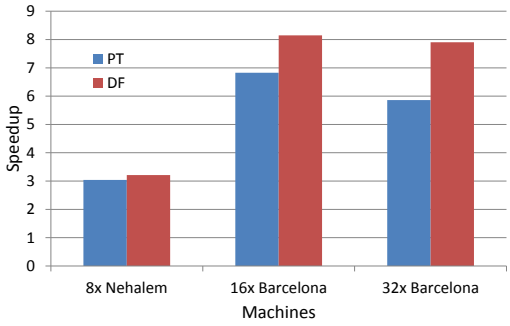

Figure 4: Harmonic mean of speedups achieved for various applications on an 8-thread, a 16-core and a 32-core machine. PT = Pthread, DF = dataflow.

2.3 Sequential Semantics

The semantics of statically-sequential programs plays a key role in accomplishing our goals, which we summarize here. Since it processes computations in the program order the model is sequentially determinate [4], which ensures in any execution of a program with the same inputs, an object is assigned the same sequence of values. This makes programs easy to reason about, and their execution predictable and repeatable, key distinctions from the statically-parallel model. It also eases coding of sequential operations such as I/O. It further permits controlling the execution while ensuring forward progress. Finally, despite requiring user-assistance, it retains much of the simplicity of the well established and understood sequential programming.

We refer readers to [23] for a detailed exposition on the model and the prototype implemented to evaluate it.

3 Continuous Adaptation of Parallel Execution for Time- and Energy-Efficiency

Utilizing resources efficiently in dynamically changing environments is going to be one of the key challenges going forward. The key to effective utilization lies in understanding the *dynamic factors* that impact the execution of the program (workload and execution environment characteristics) and appropriately *adapt* the application's parallel execution based upon those factors. While exposing too little parallelism can underutilize the resources, exposing excessive parallelism can lead to contention for resources, potentially leading to time- and energy-inefficient execution.

Statically-parallel models are ill-suited for such environments as their recipe for exposing the application's parallelism to the operating environment is written into the static program. This requires the programmer to prepare the application ahead of time to facilitate the process of adaptation in the execution environment. Several recent papers propose to dynamically vary the degree of parallelism [15, 16, 26, 28, 29, 36, 40] without any programmer involvement, but they all have several drawbacks. To summarize, they require offline analysis and learning with hints from the compiler, employ metrics and mechanisms that are tightly coupled to a particular set of resources or environment, focus mainly on array-based (data parallel) programs and require third parties to specify mechanisms for a given environment. More importantly, they cannot efficiently perform *Continuous Parallelism Adaptation (CPA)* in response to changing conditions. Doing so can require suspending, resuming or introducing computations into the environment and injudicious choice of computations in this process can lead to deadlocks, especially for programs with arbitrary dependence patterns (e.g., Cholesky decomposition). Since these proposals do not maintain any ordering information, they require complex dependence tracking algorithms to ensure that the resulting execution is indeed deadlock free, before regulating the parallelism. However, guided by the total sequential order, our model can perform CPA for the duration of the program's lifetime without requiring such complex algorithms. The model is able to ensure forward progress by dynamically constructing dependences between functions as and when they are discovered in the lexical program order and executing them in a dataflow fashion.

To determine how much parallelism to deploy at any given time, a *Goodness of Parallelism (GoP)* metric correlates the instantaneous efficiency of the program to the instantaneous degree of parallelism. As the program execution unfolds (§2.2) the model computes the GoP at periodic intervals. If the efficiency has dropped, we predict it is due to contention to some resource in the environment and alleviate the contention by decreasing the degree of parallelism deployed in the next time step. Similarly, if the efficiency has improved, we predict that the environment has more resources to offer and increase the degree of parallelism in the next time step to further improve the efficiency. If the efficiency remains the same, we predict that the environment is currently operating at its optimum and maintain the current degree of parallelism.

A current prototype of continuous parallelism adaptation, on a stock 4-core (8-thread) Intel Core i7 2600 (Sandy Bridge) workstation, computes GoP by reading energy counters and tracking tasks/instructions executed, and makes parallelism control decisions every 100 milliseconds. The net result is up to 50% higher time- and energy-efficiency over the state-of-the-art task parallel

systems across a variety of dynamic execution environments.

4 Precise-Restartable Execution

Future computer systems, built from nanoscale devices, will be unreliable [2]. Additionally, to manage resources guided by metrics such as utilization and revenue models, they will abort and/or migrate applications [30, 43]. Precise-restartable execution is desirable in such systems for a variety of objectives such as program debugging, fault-tolerant execution or to halt and resume a program at a different time, possibly on another system.

To restart a program after it has been interrupted, it is essential to identify the precise point in the program where it halted and establish the architectural state reflective of execution up to that point so that the program may be resumed without discarding all of the completed work. Doing so for statically-parallel programs is complicated. When a statically-parallel program, devoid of an order between its computations, abruptly halts, it is difficult to establish a single "point" in the program where it was interrupted. Further, parallel execution intrinsically disperses the program state among multiple processors. Hence determining a restart "point" of such a program, creating and establishing its corresponding architectural state can either require a complex system-wide online coordination, or an offline analysis that can be inconclusive [13, 19]. Researchers have proposed hardware [6, 34, 39] and/or software [13, 14, 31] schemes to implement this functionality at the cost of increased system complexity, and performance, energy and storage overheads. However, the model we propose leverages the implicit ordering in statically-sequential programs to greatly simplify the process and considerably reduce the associated overheads, as described below. (We treat all program interruptions, e.g., due to faults, as "exceptions" in this discussion.)

Parallel execution of a statically-sequential program can be made precisely-restartable after a computation excepts, if we know: (i) the order of the currently executing computations in the program, (ii) the objects they may have modified and (iii) the state of those objects prior to start of the computations. When a computation excepts its order can be used to associate a precise point in the program with the exception. Objects modified by the excepting computation and those younger to it can be restored using the state from prior to their start, causing the program state to reflect that of its sequential execution up to the exception.

To track computations and the state they may modify, we draw inspiration from the Reorder Buffer (ROB) and History Buffer mechanisms proposed to achieve precise interrupts in modern superscalar processors [38]. As we sequence through the program we track creation and completion of computations by logging them in a ROB-like structure, called the Reorder List (ROL). To maintain the history of the state a computation may alter, before it executes we make copies of the objects it may modify, i.e., its *mod set* (a user-provided set similar to the computation's write set and processed similarly to identify the objects in it). When an exception occurs the order of the computation and those younger to it is determined by their position in the ROL. Computations older to the excepting computation are allowed to complete. Program state modified by the remaining computations can be restored using the copies of their mod sets. Once the excepting condition is mitigated the program may resume from the excepting computation. Thus we realize precise-restartable parallel execution, analogous to that of sequential programs (and hence also ease other aspects of parallel systems such as debugging).

A checkpoint created by incrementally checkpointing the state after each computation successfully completes, using its mod set, represents the state of the entire program up to the completion of the computation. The program may be resumed at another time or on a different system using this checkpoint (and the identity of the last checkpointed computation). Note that in this process no cross-computation coordination is needed, conserving time and energy.

A current software prototype of the model incurs a performance overhead of 0.4% to 4.2% and energy overhead of 0% to 2.7% (when no exceptions occur) to support precise-restartability, on a 16-core AMD 8350 Opteron machine. When exceptions occur at an aggressive rate of one every second, an additional 2% to 160% time and energy overhead is incurred to recover and execute the applications (listed in §2.2) to completion.

5 Conclusion

The industry is rapidly deploying multicore processors in systems ranging from mobile devices to exascale computers. Parallel programming for these unreliable systems and their dynamically changing operating environments pose significant challenges to everyday programmers in the effort to improve productivity and to achieve error-free, efficient execution of their programs. In this paper we have argued that these challenges can be met by statically-sequential programs and their dynamically controlled dataflow execution. We showed how performance- and energy-efficient parallel execution of suitably-written sequential programs can be achieved on unreliable parallel systems. We believe this work takes a significant stride in meeting the current and emerging parallel processing challenges.

References

[1] Big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf.

[2] The international roadmap for semiconductors, 2011 edition.

[3] Posix threads programming. In *https://computing.llnl.gov/tutorials/pthreads/*, Livermore, California, USA. IEEE.

[4] Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. Applied Mathematics 14, 4 (Nov. 1966), 1390-1411.

[5] First NSF/TCPP workshop on parallel and distributed computing education. In *26th IEEE International Parallel and Distributed Processing Symposium*, IPDPS'11, Washington, DC, USA, 2011. IEEE Computer Society.

[6] R. Agarwal, P. Garg, and J. Torrellas. Rebound: scalable checkpointing for coherent shared memory. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 153–164, New York, NY, USA, 2011. ACM.

[7] M. D. Allen. *Data-Driven Decomposition of Sequential Programs for Determinate Parallel Execution*. PhD thesis, University of Wisconsin, Madison, 2010.

[8] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 85–96, New York, NY, USA, 2009. ACM.

[9] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 38:53–64, March 2010.

[10] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 81–96, New York, NY, USA, 2009. ACM.

[11] R. L. Bocchino Jr. and et al. A type and effect system for deterministic parallel java. In *OOPSLA: Object Oriented Programming, Systems, Languages and Applications*, USA, October 2009. ACM.

[12] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston. *Object-oriented analysis and design with applications, third edition*. Addison-Wesley Professional, third edition, 2007.

[13] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent advances in checkpoint/recovery systems. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 282–282, Washington, DC, USA, 2006. IEEE Computer Society.

[14] G. Bronevetsky, K. Pingali, and P. Stodghill. Experimental evaluation of application-level checkpointing for OpenMP programs. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 2–13, New York, NY, USA, 2006. ACM.

[15] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 157–166, New York, NY, USA, 2006. ACM.

[16] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 250–259, New York, NY, USA, 2008. ACM.

[17] L. Dagum and R. Menon. OpenMP: An industry-standard api for shared-memory programming. In *IEEE Computation Science and Engineering*, pages 46–55, USA, 1998. IEEE.

[18] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[19] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[20] J. G. Elytra and K. Words. Parallel data compression with bzip2.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[22] D. Gay, J. Galenson, M. Naik, and K. Yelick. Yada: Straightforward parallel programming. *Parallel Computing*, 37(9):592 – 609, 2011. ¡ce:title¿Emerging Programming Paradigms for Large-Scale Scientific Computing¡/ce:title¿.

[23] G. Gupta and G. S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th International Symposium on Microarchitecture (MICRO)*, December 2011.

[24] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 333 –334, feb. 2011.

[25] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. *SIGARCH Comput. Archit. News*, 36:265–276, June 2008.

[26] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 236–246, New York, NY, USA, 2005. ACM.

[27] E. A. Lee. The problem with threads. *Computer*, 39:33–42, May 2006.

[28] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. *SIGARCH Comput. Archit. News*, 38:270–279, June 2010.

[29] D. Li, B. R. de Supinski, M. Schulz, K. W. Cameron, and D. S. Nikolopoulos. Hybrid mpi/openmp power-aware computing. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12. IEEE, 2010.

[30] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[31] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali, and P. Stodghil. Optimizing checkpoint sizes in the C3 system. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 10 - Volume 11*, IPDPS '05, pages 226.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[32] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[33] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with lithe. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 376–387, New York, NY, USA, 2010. ACM.

[34] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 111–122, Washington, DC, USA, 2002. IEEE Computer Society.

[35] J. M. Prez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151. IEEE, 2008.

[36] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using DoPE: the degree of parallelism executive. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 26–37, New York, NY, USA, 2011. ACM.

[37] J. Reinders. *Intel Threading Building Blocks*. O'Reilly Media, Inc., 2007.

[38] J. Smith and G. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609 –1624, dec 1995.

[39] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.

[40] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multithreaded workloads on cmps. In *In Proc. 13th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 277–286, 2008.

[41] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3:54–62, September 2005.

[42] K. Vanmechelen, W. Depoorter, and J. Broeckhove. Combining futures and spot markets: A hybrid market approach to economic grid resource management. *J. Grid Comput.*, 9(1):81–94, Mar. 2011.

[43] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. *System*, (January):236–243, 2010.

[44] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *SIGARCH Comput. Archit. News*, 37:325–336, June 2009.