Implicitly Parallel Programming Models for Thousand-Core Microprocessors

Wen-mei Hwu*, Shane Ryoo*, Sain-Zee Ueng*, John H. Kelm*, Isaac Gelado†, Sam S. Stone*, Robert E. Kidd*, Sara S. Baghsorkhi*, Aqeel A. Mahesri*, Stephanie C. Tsao*, Nacho Navarro†, Steve S. Lumetta*, Matthew I. Frank*, and Sanjay J. Patel*

*Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
{hwu, sryoo, ueng, jkelm2, ssstone2, rkidd, bsadeghi, mahesri, stsao3, steve, mfrank, sjp}@crhc.uiuc.edu

†Computer Architecture Department
Universitat Politecnica de Catalunya (UPC)
{igelado, nacho}@ac.upc.edu

ABSTRACT

This paper argues for an implicitly parallel programming model for many-core microprocessors, and provides initial technical approaches towards this goal. In an implicitly parallel programming model, programmers maximize algorithm-level parallelism, express their parallel algorithms by asserting high-level properties on top of a traditional sequential programming language, and rely on parallelizing compilers and hardware support to perform parallel execution under the hood. In such a model, compilers and related tools require much more advanced program analysis capabilities and programmer assertions than what are currently available so that a comprehensive understanding of the input program's concurrency can be derived. Such an understanding is then used to drive automatic or interactive parallel code generation tools for a diverse set of parallel hardware organizations. The chip-level architecture and hardware should maintain parallel execution state in such a way that a strictly sequential execution state can always be derived for the purpose of verifying and debugging the program. We argue that implicitly parallel programming models are critical for addressing the software development crises and software scalability challenges for many-core microprocessors.

Categories and Subject Descriptors: D.1.3 [Software]: Programming Techniques—*Concurrent Programming*

General Terms: Design, Human Factors, Languages

Keywords: Parallel Programming

DAC 2007, June 4-8, 2007, San Diego, California, USA

1. INTRODUCTION

The design of microprocessors, general-purpose and embedded alike, is converging to that based on multiple processor cores. These microprocessors vary in the number and size of their processor cores, depending on market requirements and fabrication capabilities. It will be tempting for vendors to impose explicitly parallel programming models for these platforms. The higher the level of hardware parallelism, the stronger the temptation will be. In this paper, we take a potentially unpopular position and argue that explicitly parallel programming will be counterproductive for the vast majority of programmers in the long run. Instead, implicitly parallel programming models properly supported by compile tools and hardware will be the preferred approach.

It is important to note that we are not advocating the use of automatic parallelization of programs based on sequential algorithms in general. The cases where a smart compiler can recognize that an equivalent, more parallel algorithm is available for a particular computation are limited. We believe that programmers are better at understanding the trade-offs necessary to choose a more parallel but not necessarily completely equivalent algorithm.

Another important clarification is that we are not advocating the general use of legacy code as input to the tool chain. Although the techniques and tools proposed for the implicitly parallel programming models will likely work to a certain extent on legacy code, we expect that they will deliver their full benefit to code developed or revised under the proposed methodology and flow for two reasons. First of all, legacy code is often developed using sequential algorithms and needs to be redeveloped with more parallel algorithms. Second, legacy code often contains coding styles and unexpressed high-level assumptions that makes it extremely difficult for the tool chain to derive a correct, deep understanding of the high-level properties and assumptions of the program. We believe that the legacy code base must be revised or redeveloped not only to use many-core systems, but also to enable future software engineering tools to help increase the reliability of the code base [1].

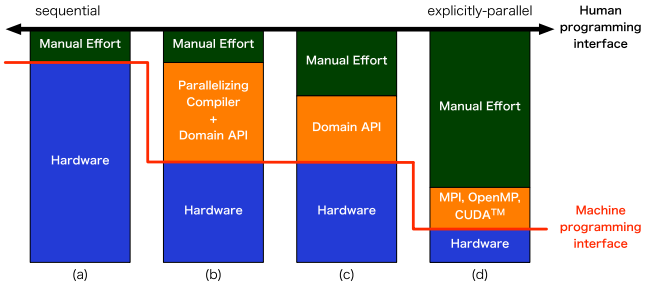


Figure 1. Current Programming Models of Major Semiconductor Programmable Platforms. Ranges from sequential (a) to explicitly parallel (d) models.

It is also important to point out that explicitly parallel programming models have been used by expert programmers for decades and will continue to be used in specialized situations. This is similar to how assembly-level programming is still used today when code is performance-critical and resource usage must be tightly controlled. In addition, when the set of executing applications is limited, it may be more cost-effective to reduce hardware design cost and require explicitly parallel software for the platform. However, just as high-level programming paradigms enabled the explosive growth of the software industry, we envision implicitly parallel programming models to have a similar effect.

We believe that domain- and processing model-specific languages [19, 21] with specialized compilers are near-optimal choices when developing applications that fit the domain or model. We do not consider them to be explicitly parallel programming models. For example, Shah et al. [19] are able to achieve both high productivity and execution efficiency with this language-application matching. However, not all domains will be of sufficient size or maturity to justify development of specialized languages.

What we do advocate is that programmers should express their parallel algorithms in a "canonical form," where a sequential ordering exists among all parallel units of execution. This sequentially-ordered representation will be in the form of a traditional sequential language such as C/C++. We refer to this representation as an implicitly parallel programming model in the sense that systematic, efficient code analysis and transformations by the compiler and the underlying hardware can fully expose, map, and exploit the concurrency in the algorithms. We argue that the sequentially-ordered representation allows a more tractable interface for testing, debugging, and supporting parallel applications while allowing the full exploitation of algorithmic parallelism. Applications based on these implicitly parallel programming models will more readily take advantage of each additional step of Moore's Law [12] scaling to thousands of processor cores than their explicitly parallel counterparts.

2. PREVIOUSLY SUCCESSFUL MODELS

Microprocessors have long employed parallel processing at the hardware level. They have, however, taken one of the three approaches shown in Figures 1(a) through (c) to hide the complexity of parallel execution from programmers. The top horizontal line in Figure 1 depicts the interface to human, high-level programmers whereas the middle line depicts the machine-level programming interface to compilers and debuggers.

Figure 1(a) shows the approach where the instruction-set architecture (ISA) and its hardware implementation completely hide the complexity of parallel execution activities from both the human programmer and the compiler. This is used by superscalar processors, such as the Pentium® 4 processor [3], where the fact that the hardware can execute instructions in parallel and out of their program order is hidden by a sequential retirement mechanism; the execution state exposed through debuggers and exception handlers to human programmers is completely that of sequential execution. Thus, programmers never deal with any complexity or incorrect execution results due to parallel execution. This is the dominant model of managing parallel execution in general-purpose microprocessors today, accounting for hundreds of millions of units per year. It is also the programming model for the vast majority of programmers today.

Figure 1(b) illustrates a model where parallel execution is exposed at the machine programming level to compilers and object-level tools, but not to human programmers. The most prominent microprocessors based on this model are VLIW processors in the embedded processor domain [11, 23] and EPIC processors in the server processor domain [9]. In these models, programmers see traditional sequential programming models such as C/C++. They need to recompile their source, and in some cases rewrite their source code so that algorithms with more inherent parallelism are used and coding style and constructs better match those that can be analyzed and manipulated by the parallelizing compiler. Critical math libraries are often developed by vendor engineers at the machine programming level to further enhance the execution efficiency of these processors, shown as the domain application programming interface (API) in Figure 1(b). Note that the complexities of parallel execution are exposed to human programmers whenever they need to use debuggers, since the hardware does not maintain sequential state. It is well known that the acceptance of these processors in the general-purpose market has been hampered by the need to recompile source code and to deal with the parallel execution complexities exposed by the debuggers when source program or compiler bugs arise.

Figure 1(c) shows a model that has been successfully used in specialized market domains such as graphics processing. In this model, the complexities of parallel execution in hardware are hidden from application programmers via a layer of domain-specific API functions. These API functions perform substantial computation tasks such as rendering a set of 3-D objects using the parallel execution capabilities of the underlying hardware. The application programmers, however, write sequential code that invokes these API functions. Although graphics processing units (GPU) have long employed multiple dozens of processor cores, few application programs in the graphics domain have had to deal directly with parallel execution.

All three models have been successfully used in their respective target markets by carefully hiding the complexity of parallel execution from their applications programmers. These arrangements are, however, being changed in the strategies adopted by virtually all semiconductor vendors. Figure 1(d) illustrates examples of the contemporary strategies adopted by vendors. In the general-purpose domain, MPI [10] and threading [7, 20] are currently the main programming models for using many-core microprocessors to exploit parallel execution within an application. The

new CUDA programming model [13] from NVIDIA allows programmers to write massively threaded parallel programs for GPUs without dealing with domain-specific API functions. Multiple Instruction Stream Processing (MISP) [6] is proposing to move some of the complexity of thread scheduling to the user-level above the OS. We argue that while these programming models could serve as good machine-level programming models targeted by parallelizing compilers, they are unlikely to be cost-effective for the vast majority of application programmers.

3. COST OF PARALLEL PROGRAMMING

In order to appreciate the need for implicitly parallel programming models, one must first understand the reasons why software development using explicitly parallel programming models is an expensive proposition. Explicitly parallel programming models are not new. The high-performance scientific computing community has been developing applications based on models such as OpenMP [14] and MPI for more than two decades. The transaction processing community has also been developing explicitly parallel application programs for a long time. The complexity of parallel programming is well documented in the literature [4, 8, 15].

In order to develop an explicitly parallel program, programmers must understand the concurrency of the algorithms, determine the granularity of parallel execution, set up data structures to allow correct parallel execution, and rewrite the program to execute in parallel. However, to achieve efficient implementation of parallel applications, programmers must also understand the underlying hardware and perform optimizations whose effects are often determined by tedious and error-prone experimentation.

There are additional issues that further increase the cost of explicitly parallel programming models. First, scaling of software performance is difficult. Moving software from one hardware platform to a successor with more parallelism often requires the repetition of the experimentation process. The requirement of general purpose software to be capable of running on a plethora of platforms from different vendors further exacerbates this issue. Second, explicitly parallel code does not compose well. In general, one cannot use parallel code components and build large parallel systems and expect to have correct and efficient execution. Furthermore, one must verify and debug these programs in the presence of non-deterministic execution states that vary according to execution timing. These complexities have limited the production use of parallel programming models to a very small fraction of the applications programming community.

With the complexity of parallel programming in mind, let us now put the implications of the various models in Figure 1 into the perspective of the overall industry. Figure 2(a) illustrates the traditional, sequential interface between hardware and software [18]. The bottom triangle represents the semiconductor industry with its hardware and software tool products. The top triangle represents the application development industry including independent software vendors, information technology service providers, and information technology consulting firms. Note that there is a very narrow interface between the top and bottom triangles, symbolizing the fact that very few details and complexities are exposed across the interface. This situation corresponds to the

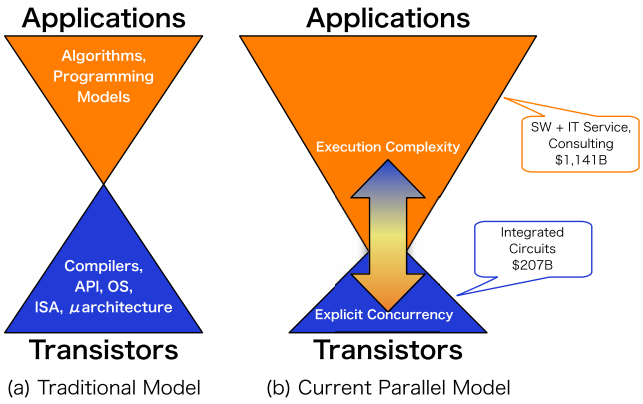


Figure 2. Cost and Complexity Exposed to Programmers

models shown in Figure 1(a)-(c): application programmers know nothing or very little about the complexity of parallel hardware execution. It is also true that in this traditional model of interface, the programmers inform the tools and hardware of nothing or very little about the nature of the computation being performed.

Figure 2(b) shows the interface model currently being adopted by the semiconductor industry. The interface has been widened, exposing more information and complexity between the two industries. From the bottom-up direction, the application programmers are now exposed to the complexities of parallel execution. Conversely, tools and hardware are now required to deal with the explicitly-expressed concurrency in the applications they process. The problem is that such an exposure of complexity creates a major roadblock for the future growth of the semiconductor industry.

The triangles in Figure 2(b) are sized according to the estimated sizes of the two industries. The 2005 revenue of the global integrated circuit market is estimated to be $207B whereas the revenue of the global applications industry is estimated at $1,141B [5] and has generally grown at a faster rate. This size ratio means that effort by the semiconductor industry to increase productivity and value of applications is magnified by about 5 times. This positive magnification has been an important ingredient of the positive cycle feeding the growth of the semiconductor industry.

The problem with pushing explicitly parallel programming models on application developers is that it is an exposure of complexity and elevation of costs to the applications industry as a result of lack of effort, innovation, or support by the semiconductor industry. It diminishes the positive cycle mentioned before and takes away one of the fundamental driving forces of the growth of the semiconductor industry.

4. IMPLICITLY PARALLEL PROGRAMMING

The practice of implicitly parallel programming is not new. The high-performance scientific community has been working on parallelizing FORTRAN compilers for a long time. The architecture community has also worked on speculative multi-threading techniques to maintain sequential execution state while using multiple processors to speed up a sequential thread. In this section, we will present more details of the proposed implicitly parallel programming model and its relationship to the previous work.

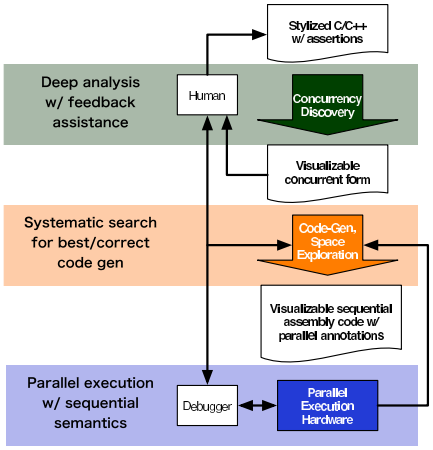


Figure 3. Implicitly Parallel Programming Model Flow

Figure 3 shows the tool and execution flow of our proposed implicitly parallel programming model. In this model, parallel algorithms are expressed in C/++/# and Java without using explicitly parallel libraries or parallel language constructs. The programmer can also provide assertions about the high-level properties and invariants of the application software. Examples include marking user custom memory allocation pools to enable heap sensitive pointer analysis and memory access behavior analysis of library functions whose source code may be missing from the compilation process. These are in contrast to programmer-inserted directives [14] used by previous parallelizing compilers to drive transformations in the absence of compiler analyses.

We have mentioned the need to use parallel algorithms rather than serial ones. Figure 4 shows pseudo-code for two possible implementations of the motion estimation loop for an MPEG-4/H.263 encoder, with corner/boundary cases omitted for brevity. In both algorithms, `GetMatch`, which has no memory side effects, searches for a good fit for a 16x16 pixel partial image, or *macroblock*, `cur_frame[i]` within the previous frame `prev_frame`, with the search guided by an estimated offset vector `guess`. The algorithms differ only in their methods for generating the guess vector. We will use this example to demonstrate the types of algorithmic tradeoffs a programmer must consider when using an implicitly parallel programming model.

In Figure 4(a), the motion estimator obtains a guess from the previous macroblock in horizontal scan order. The theory underlying this choice is that objects tend to be larger than a single macroblock and the pieces of the object move with the same vector. This technique is an example of adaptive search and provides a reasonably accurate and efficient result on a uniprocessor system. However, this particular adaptive search has poor parallelism between macroblocks. The previous macroblock in the chain must be completely processed before a macroblock can obtain its guess vector and begin processing. Thus, this algorithm is not preferred for execution on a many-core system.

Figure 4(b) shows another adaptive motion estimation algorithm that obtains guess vectors from the corresponding macroblock in the previous frame, rather than the previous macroblock in the current frame. The theory underlying this guess is that objects tend to have inertia and do not have radical shifts in velocity between frames because the time slices are small. This search has far better parallelism between macroblocks: macroblocks in a frame can be processed independently because there is a dependence only between a macroblock in the current frame and the corresponding macroblock in the previous frame, rather than between the current macroblock and the previous macroblock in the same frame. Moreover, removing this dependence would not expose more parallelism for MPEG-4 motion estimation, because the image data for the previous frame is inherently required to create the new frame. Consequently, this choice of guess vector fits well with the existing dependences in the overall encoding algorithm.

Note that these two algorithms do not produce exactly the same result, which has ramifications on image quality, encoded video size, and more. This trade-off between available parallelism and various other desirable characteristics is exactly where human innovation and problem domain knowledge is required. We believe that no proposed tool or programming language should make this decision.

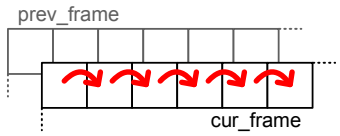
4.1 Concurrency Discovery

The Concurrency Discovery module in the flow is responsible for rediscovery of concurrency from the sequential language representation of the parallel algorithms. Although this sounds like redundant work on the surface, it is a worthwhile effort that reduces the tedious and error-prone work otherwise performed by the programmer. The techniques comprising this module include, for example, advanced pointer and memory data flow analyses that construct the program concurrency and dependence information from the bottom up. The fundamental approach is to summarize all of the memory locations that each code region can access and use the information to determine if two code regions can be executed in parallel without interference. It also identifies execution dependences that can be removed by data replication.

Referring again to the motion estimation algorithms of Figure 4, the Concurrency Discovery module detects that the algorithm in part (a) contains a loop-carried dependence; therefore, it cannot be parallelized. The loop in part (b) contains no such loop-carried dependence. However, if the current and previous frames were allocated with a single call to `malloc`, the compiler may yet be unable to determine that the elements of the current and previous frames are not aliased. In a subtle case like this, the module needs to indicate to the programmer what dependences are holding back concurrency discovery, i.e. the feedback loop in Figure 3. The programmer, who understands the high-level parallelism, can then assert that the two frames do not overlap (`user_assert`), allowing the Concurrency Discovery module to determine that the loop is parallelizable.

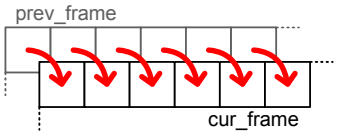
Previous work [17] has discussed the battery of analyses that can be used to expose the parallelism in an MPEG-4 encoder, including a context sensitive, interprocedural pointer analysis capable of distinguishing among heap objects allocated from different `malloc` call chains, the Omega test [16], and accurate analysis of non-affine index expressions for cross-iteration dependences. Mileage can vary significantly across different types of analysis techniques. An automatic parallelizing compiler equipped with deeper analysis capabilities can be much more effective than another with more traditional, weaker versions of the same general categories of analysis techniques. Work also exists for profile-based discovery of parallelism, which can provide additional information when analysis information is incomplete [24].

```
for (i = 0; i < num_blocks; i++)
  Vector guess = i ?
    (cur_frame[i-1].best_vector) : (0,0);
  cur_frame[i].best_vector =
    GetMatch(cur_frame[i], prev_frame, i, guess);
```



(a) Guess vectors are obtained from the previous macroblock.

```
user_assert(not_overlapping(cur_frame, prev_frame));
for (i = 0; i < num_blocks; i++)
  Vector guess = prev_frame[i].best_vector;
  cur_frame[i].best_vector =
    GetMatch(cur_frame[i], prev_frame, i, guess);
```



(b) Guess vectors are obtained from the corresponding macroblock in the previous frame.

Figure 4. MPEG-4/H.263 Encoder Motion Estimation Example and Dependence Visualization. The algorithm on the left is sequential because every iteration depends on the previous macroblock. The loop on the right can be parallelized because this dependence has been removed.

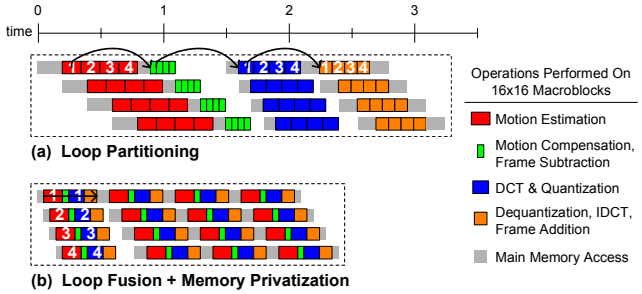


Figure 5. MPEG-4/H.263 Alternative Code Generation Strategies

4.2 Code-Gen Space Exploration

In an application, there are usually multiple ways to exploit concurrency; the choice often depends upon the underlying hardware organization. Figure 5 shows two example strategies for executing the MPEG-4/H.263 encoder with execution timing of each core shown in a horizontal time frame.

In Figure 5(a), iterations of the motion estimation outer loop are executed in parallel, followed by parallel execution of iterations of the loops for motion compensation, DCT/quantization, and dequantization/IDCT. Each loop body contains multiple levels of inner loops and function calls. The advantage of this approach, loop partitioning, is that it is simple and requires very little synchronization among processor cores. The disadvantage of this strategy is that it tends to make poor use of cache memories. The information generated by an iteration of the motion estimation loop is wiped out of the cache by subsequent iterations before it can be consumed by the motion compensation loop.

The strategy in Figure 5(b) solves the data cache efficiency problem by fusing the four outer loops into a single large loop body. In each iteration of the fused loop, the motion estimation result is immediately consumed by motion compensation. This eliminates unnecessary data movement in and out of the data cache. However, the fused loop also significantly increases the instruction working set per processor. In processors with small instruction caches, this strategy can actually incur more cache misses than that in Figure 5(a).

This illustrates the cost of explicitly parallel programming models: different underlying hardware organizations often require different parallelization strategies to achieve good performance. Instead of performing transformations by hand, we propose that programmers write a single source code and let the compiler determine the appropriate code generation strategy, whether by trial and error [2] or by realistic performance estimates [22], and perform the code transformations to achieve it. These actions are performed by the Code-Gen Space Exploration module in Figure 3.

4.3 Parallel Execution Hardware

The execution hardware in Figure 3 has a wide spectrum of design alternatives. In the general-purpose domain, we argue that the hardware should be capable of producing a strictly sequential execution state in the presence of any exceptions or breakpoints. This can be achieved by a variety of combinations of compiler and hardware support. In one embodiment, the machine-level executable binary consists of two parts, sequential code and parallel annotations. The sequential code can be correctly executed by a traditional single-threaded processor. It defines the sequential semantics of the executable code. The parallel annotations specify the modifications to be made to the sequential code by the hardware in order to execute the program in parallel. We envision the annotations to be similar to OpenMP, but treated by the hardware as hints, rather than directives. During normal execution, the parallel annotation directs the hardware to execute the code in parallel. The hardware enforces execution rules so that one can always produce an equivalent sequential execution state on a need basis. This requirement necessitates extra hardware support and may result in situations where a particular form of concurrency may not be exploited due to this requirement.

For some algorithms, the system can easily provide sequential state for exception handling even with minimal hardware support. For the example above, the motion estimator does not overwrite the input arrays during parallel execution and writes to each output array element at most once. Consider an exception that arises while computing `cur_frame[37].best_vector`. A simple, naive approach would be to let all computation finish before the excepting core reports the corresponding program counter value and the values of `i` and `guess`. Given those values, a "sequential controller" could then synthesize a sequential state, thereby allowing the exception handler to see all elements of `cur_frame` before `cur_frame[37]` correctly. Although the handler would also see elements after `cur_frame[37]` computed too early, the exception handlers are only concerned with the values generated before the exception point.

Because the input `prev_frame` array is not overwritten, execution can simply resume after the exception handler with the sequential state. Note that there are many other potential strategies and specific mechanisms to support more sophisticated forms of parallelism and to reconstruct the sequential state more efficiently. This particular strategy is

given to illustrate the basic ideas of our proposed machine-level programming models.

If we consider the other end of the spectrum, the embedded computing domain, this level of hardware support is probably undesirable due to the associated cost and smaller set of applications. In this case, the hardware can provide an explicitly parallel, multi-threaded interface to the compiler to allow for low-cost, low-power implementations. Although programmers still program in sequential languages, the compiler generates explicitly parallel machine-level code. This model exposes the complexity of parallel execution through debuggers to the human programmers.

5. CONCLUSION

In this paper, we presented the main reasons why implicitly parallel programming models are critical to address the software crisis of microprocessors. Although it is extremely tempting for the semiconductor industry to require programmers to write explicitly parallel programs, we argue that it will become a major roadblock to the future growth of the semiconductor industry. Instead, we present an implicitly parallel programming model that requires much more compiler and hardware capabilities than are currently available. One may ask why we believe that such capabilities will come into being after thirty years of futile attempts. Our answer is that two major new developments have occurred in the meantime. First, parallelizing compilers with much more advanced, robust analysis capabilities have been successfully commercialized for instruction-level parallel processing systems such as the Intel Itanium® and TI's C60. Solid progress has been made in the area of concurrency detection and code generation for parallelism. Much progress has also been made on program analysis on large applications for software engineering purposes at companies such as Microsoft. The technical foundation of parallelizing compilation has advanced greatly without much general recognition. The second development is that the amount of hardware resources that can be spent on maintaining execution state and checking equivalence to sequential execution has increased in a dramatic way in the past decade. Based on these observations, we believe that implicitly parallel programming models will begin to be widely adopted by application developers in the next few years.

6. ACKNOWLEDGMENTS

The authors would like to thank Kurt Keutzer, Scott Mahlke, Marc Snir, Brian Deitrich, Jan Rabaey, and all IMPACT research group members for their valuable comments. This research has been supported by the Gigascale Systems Research Center (GSRC), a generous gift from Microsoft under the Phoenix program, and an equipment grant from the National Science Foundation under NSF CNS 05-51665.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. of Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, volume 3362 of *LNCS*, pages 49–69. Springer, March 2004.

[2] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation*, October 1998.

[3] D. Carmean. The Pentium 4 processor. Hot Chips 13, Stanford University, Palo Alto, CA, August 2001.

[4] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1997.

[5] Datamonitor, 2006. http://www.datamonitor.com.

[6] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 114–127, June 2006.

[7] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, 2001.

[8] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[9] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE MICRO*, 23(2):44–55, March 2003.

[10] Message Passing Interface Forum. MPI-2: extensions to the message-passing interface, July 1997.

[11] N. Mitchell. Philips TriMedia: A digital media convergence platform. In *Proceedings of WESCON*, pages 56–60, November 1997.

[12] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.

[13] NVIDIA Corporation. *CUDA Programming Guide*, February 2007.

[14] OpenMP Architecture Review Board. OpenMP application program interface, May 2005.

[15] G. Pfister. *In Search of Clusters*. Prentice Hall, 1997.

[16] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing 1991*, pages 4–13, November 1991.

[17] S. Ryoo, S.-Z. Ueng, C. I. Rodrigues, R. E. Kidd, M. I. Frank, and W. W. Hwu. Automatic Discovery of Coarse-Grained Parallelism in Media Applications. *Transactions on HiPEAC I, LNCS 4050*, 2007.

[18] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[19] N. Shah, W. Plishker, K. Ravindran, and K. Keutzer. NP-Click: A productive software development approach for network processors. *IEEE Micro*, 24(5):42–54, September/October 2004.

[20] Sun Microsystems. Java. http://java.sun.com.

[21] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[22] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[23] J. Turley and H. Hakkarainen. TI's new 'C6x DSP screams at 1,600 MIPS. *Microprocessor Report*, 11(2):14–17, February 1997.

[24] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 25–36, February 2007.