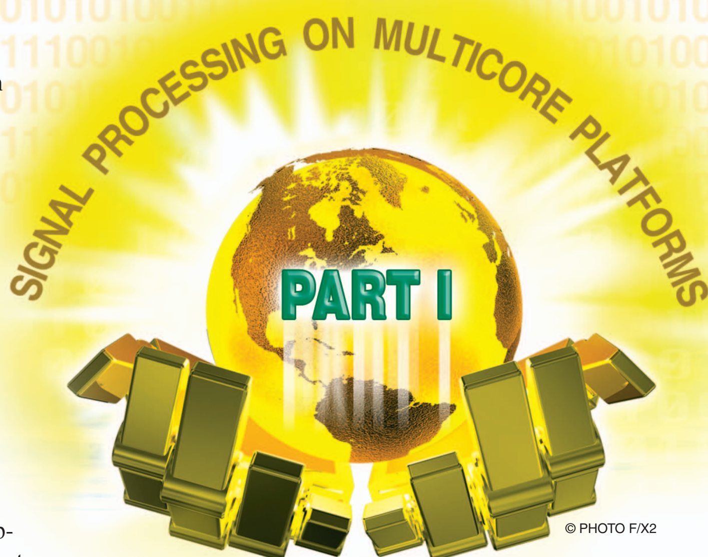
Mojtaba Mehrara, Thomas Jablin, Dan Upton,
David August, Kim Hazelwood, and Scott Mahlke

Multicore Compilation Strategies and Challenges

An overview of parallelism and compiler technology



© PHOTO F/X2

To overcome challenges stemming from high power densities and thermal hot spots in microprocessors, multicore computing platforms have emerged as the ubiquitous computing platform from servers down through embedded systems. Unfortunately, providing multiple cores does not directly translate into increased performance or better energy efficiency for most applications. The burden is placed on software developers and tools to find and exploit coarse-grain parallelism to effectively make use of the abundance of computing resources provided by these systems. Concurrent applications are much more complex to develop than their single-threaded ancestors, thus software development tools will be critical to help programmers create both high performance and correct software. This article provides an overview of parallelism and compiler technology to help the community understand the software development challenges and opportunities for multicore signal processors.

INTRODUCTION

For more than four decades, the semiconductor industry has depended on Moore's law to deliver consistent application performance gains through the multiplicative effects of increased transistor counts and higher clock frequencies. However, power dissipation and thermal constraints have emerged as dominant design issues and forced architects away from relying on increasing clock frequency to improve performance. Exponential growth in transistor counts still remains intact and a powerful tool to improve performance, though the paradigm through which performance is perceived has shifted. Performance is now based on throughput and efficiency, utilizing multiple cores performing computation in parallel to complete a larger volume of work in a shorter period of time. These multicore systems have become the industry standard from high-end servers down through desktops and gaming platforms. Example systems include the Texas Instruments (TI) TMS320C6474 that has three eight-wide C64x very long instruction word (VLIW) cores, the Sun UltraSparc T1 that has eight cores, the Sony/Toshiba/IBM Cell processor that consists of nine cores, the NVIDIA GeForce 8800 GTX that contains 16 streaming multiprocessors,

Digital Object Identifier 10.1109/MSP.2009.934117

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

UNFORTUNATELY, PROVIDING MULTIPLE CORES DOES NOT DIRECTLY TRANSLATE INTO PERFORMANCE.

each with eight processing units, and the Cisco CRS-1 Metro router that utilizes 192 Tensilica processors.

Embedded platforms, on the other hand, have long consisted of multiprocessor systems-on-chip (MPSoC), but they are typically heterogeneous combinations of hardwired accelerators and standalone processors to meet cost, performance, and power requirements. We believe embedded systems will follow the trend of servers, desktops, and graphics processors of adopting multicores as the standard platform. This trend is evident by the new digital signal processor (DSP) systems proposed by companies for wireless baseband processing that all consist of multiple single-instruction, multiple data (SIMD) processors: Ardbeg by ARM, MuSIC by Infineon, NXP by Philips, and Sandblaster by Sandbridge. Multicores provide a systematic way to scale performance in the face of relatively stagnant clock rates without skyrocketing power consumption and design complexity. Further, multicores have an important advantage over current MPSoC systems—programmability. A programmable solution offers the opportunity for a single platform to support multiple applications and even multiple standards within each application domain. Finally, programmability provides faster time to market as hardware and software development can proceed in parallel, the ability to fix bugs and add features after manufacturing, and higher chip volumes as a single platform can support a family of mobile devices.

Unfortunately, providing multiple cores does not directly translate into performance. With multicores, the industry has already fallen short of the decades old single-thread performance growth trend, and the trend toward simpler cores means performance might even degrade. Not only do sequential codes suffer, but multithreaded programs may degrade due to smaller caches per core, limited memory bandwidth, and lower single-thread performance.

Many new languages have been proposed to ease the burden of writing parallel programs, including Atomos, Cilk, and StreamIt. Despite these and more than 150 other parallel languages, the effort involved in creating correct and efficient parallel programs is still far more substantial than writing the equivalent single-threaded version. Developers must be trained to program and debug their applications with the additional concerns of deadlock, livelock, and race conditions. Converting an existing single-threaded application is often more challenging, as it may not have been developed to be easily parallelized in the first place. Extracting the fine-grained parallelism necessary for efficient use of multicore systems is not only tedious, but it is a continuous cost as the machine-specific partitioning lack portability and forward performance compatibility. And finally, programmers cannot (and should not) be aware of the details of the hardware resources available at run time in the face of varying system load.

The difficulty and complexity of programming for multicores inevitably pushes programmers, particularly those in the embedded community who have long relied on hand assembly code or low-level C code, to rely more heavily on tools, such as compilers and run-time optimizers. These tools automatically extract threads, perform machine-dependent mappings of threads, and manage run-time program execution. This approach avoids the pitfalls resulting from exposing the multicore problem up through the entire software stack and allows the programmer to focus on problem solving and correctness. This article provides an overview of multicore compilers and their future for the signal processing community.

FORMS OF PARALLELISM

Traditional DSPs adopted application-specific units and instruction set extensions to accelerate signal processing algorithms. The major role of compilers in traditional DSPs was minimizing code size. High performance code was generally written by hand or with heavy use of intrinsics. Due to the characteristics of DSP applications, there are many ways to achieve higher performance and increase computational efficiency through exploiting different forms of parallelism [1]. In this section, we explore various forms of parallelism available in signal processing applications.

INSTRUCTION-LEVEL PARALLELISM

With instruction-level parallelism (ILP), multiple independent assembly instructions are executed in the processor at the same cycle [Figure 1(a)]. Superscalar, general-purpose processors exploit ILP by issuing multiple independent instructions in each cycle that are dynamically identified by the hardware. However, these architectures incur a considerable amount of complexity and power consumption, which makes them less appealing for embedded systems.

The emergence of VLIW architectures introduced new opportunities for exploiting ILP in signal processing applications. In VLIW architectures, independent instructions are packed into a single large instruction word and issued in parallel each cycle. Compilers perform the important task of identifying ILP opportunities and scheduling independent instructions for execution in the same cycle. To create an instruction schedule, the exact underlying architecture (e.g., number and latency of execution units, etc.) must be known at compile time. The TI C6x is the most well-known family of VLIW DSPs that can issue eight operations each cycle [2]. TI compilers combine sophisticated optimization and scheduling techniques to identify high degrees of ILP, particularly in loop-intensive code.

DATA-LEVEL PARALLELISM

The most dominant form of parallelism in signal processing is data-level parallelism (DLP) wherein the same instruction is performed on different pieces of data in parallel [Figure 1(b)]. DLP originated in the form of SIMD and vector computation models. From the mid-to-late 1990s, several major vendors of

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

general-purpose processors included short vector SIMD extensions to their instruction sets. Some examples are Intel SSE and SSE2, AMD 3DNow, ARM NEON, and Motorola AltiVec (which was implemented on several PowerPC processors including G4, G5, and POWER6). The introduction of these extensions brought the SIMD model to the forefront of DSP technology. These extensions enabled exploitation of limited DLP by inserting small data types in large registers and using SIMD instructions to perform the same operation on all data within these registers in parallel. Vectoring compilers exist to make automatic use of SIMD extensions, but in most cases the burden is still on programmers to identify DLP.

THE MOST DOMINANT FORM OF PARALLELISM IN SIGNAL PROCESSING IS DATA-LEVEL PARALLELISM.

Modern graphics processing units (GPUs) adopt wider SIMD implementations (128–256 b). One of the latest examples is the NVIDIA's Compute Unified Device Architecture (CUDA) framework [3]. CUDA is a general-purpose parallel computing architecture that consists of the CUDA instruction set and the compute engine in the GPU. It provides a small set of extensions to the C programming language, which enables straightforward implementation of parallel algorithms on the GPU. CUDA also supports scheduling the computation between the CPU and GPU, such that serial portions of applications run on the CPU and parallel portions are mapped to the GPU. Since the public release of CUDA framework in 2007, it has been shown that many compute-intensive signal processing and multimedia applications observe remarkable speedups using this framework [4], [5]. NVIDIA provides a compiler system (nvcc) as a part of the framework.

The Intel Larrabee [6] and IBM Cell [7] processors are examples of multicore architectures augmented by wide SIMD processing units. Scheduling is performed entirely in software, and the native programming model supports implementing various parallel applications even with irregular data structures. The Larrabee and Cell C/C++ compilers provide autovectorization, and developers may program SIMD units with C++ vector intrinsics or inline assembly code.

LOOP-LEVEL PARALLELISM

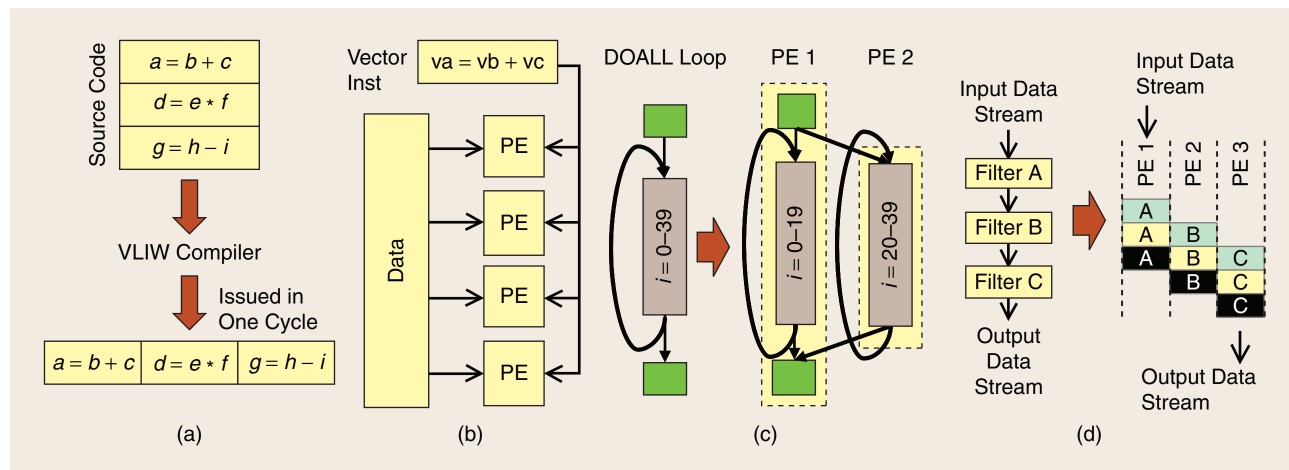
Loop-level parallelism (LLP) is one of the popular parallelization methods in the scientific computing community. In this form of parallelization, independent iterations of the same loop are executed in parallel on different processors [Figure 1(c)]. There has been a long history of automatic parallelization efforts in compilers, e.g., Polaris from Illinois, SUIF from Stanford, and Parascope from Rice. These techniques mainly target counted loops that manipulate array accesses with affine indices.

Compilers play an important role in identifying and extracting parallelism from loops. Memory dependence analysis can be precisely performed and many loops with independent iterations (DOALL loops) can be statically identified at compile time. Furthermore, compiler transformations can be used to expose more parallelism from the loops that have parallelism lurking beneath the surface in the code [8]. One example of these transformations is loop interchange, which is done on a set of nested loops. When parallelism exists in the inner nest, the compiler can exchange the loop with the outermost loop and thereby maximize the exploited parallelism.

In addition to automatic parallelization of loops, the programmer can specify parallel loops by using explicit parallel programming constructs such as the OpenMP application programming interface. Using these constructs, the application is executed serially until it reaches a parallel loop. Subsequently, several threads are spawned to execute different iterations of the loop, and when the loop execution is completed, parallel threads are joined and the program continues with sequential execution.

PIPELINE PARALLELISM—STREAM PROGRAMMING

In the pipeline parallelism model (also called stream programming, or streaming model), the application is decomposed into a series of stages. Each stage performs partial processing on a



[FIG1] Various forms of exploiting parallelism: (a) ILP, (b) DLP, (c) LLP, and (d) the streaming model called pipeline parallelism.

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

set of data and forwards it to the next stage for processing, and starts working on the next set of data. In this scenario, pipeline stages can run in parallel while working on different chunks of data [Figure 1(d)].

PARALLELISM CAN BE THROTTLED, REMAPPED, OR REORGANIZED TO TAKE ADVANTAGE OF SPECIFIC EXECUTION CIRCUMSTANCES KNOWN ONLY AT RUN TIME.

Stream languages are mainly motivated by the application style used in signal and image processing, graphics, networking, and other media processing domains. They enable the explicit specification of producer-consumer parallelism between coarse grain units of computation. Examples of stream languages are StreamIt, Brook, CUDA, SPUR, Cg, Baker, and Spidle.

For instance, StreamIt [9] represents a program as a set of autonomous actors (called filters), which are similar to Java classes. These filters communicate through first-in, first-out (FIFO) data channels. StreamIt implements a synchronous dataflow model in which the number of data samples produced and consumed by each filter are specified a priori. Each filter has a separate instruction stream and an independent address space, thus all dependencies between filters are made explicit through the communication channels. A large number of signal processing applications have been implemented as a set of StreamIt benchmarks. Examples include fast Fourier transform (FFT), discrete cosine transform (DCT) and MPEG decoder/encoder.

Stream programs generally contain an abundance of explicit parallelism. However, the central challenge is obtaining an efficient mapping onto the target architecture. The gains obtained through parallel execution can often be overshadowed by the costs of communication and synchronization. Resource limitations of the system must also be carefully modeled during the mapping process to avoid stalls. Resource limitations include finite processing capability, limited memory associated with each processing element, interconnect bandwidth, and direct memory access (DMA) latency.

ROLE OF THE COMPILER

Signal processing applications have been traditionally developed in the form of sequential algorithms. These implementations later evolved to more complex domain-specific vectorized computations that have been adopted extensively in DSPs. Explicit parallel programming models, such as stream programming and OpenMP, provide the programmer with convenient constructs to expose coarse-grain parallelism. From the compiler perspective, this provides a spectrum of programming models in the signal processing domain that must be dealt with—from implicit parallelism that must be discovered in sequential C and C++ implementations to explicit parallelism that must be effectively managed to deliver performance on the target platform. The compiler's responsibilities can be broken down into two major categories based on when the action occurs: static, which occurs offline, and dynamic, which occurs online.

In the static category, one of the major focuses is identifying parallelism: implicit parallelism in seemingly sequential applications and finding more parallelism in explicitly parallelism applications, e.g., identifying LLP inside individual filters in a streaming application. These tasks are traditionally done through a combination of compiler analyses and transformations to understand the detailed memory access behavior of the application in concert with a set of transformations to expose more parallelism. With parallelism in hand, the static compiler is employed to efficiently answer the questions of how, when, and where should the parallel segments of the applications be mapped onto the underlying hardware. The main objective is to make intelligent decisions based on the available resources and the relative communication and synchronization costs.

While most signal processing software developers think of compilers exclusively in the static sense, multicore compilers will likely also have a dynamic component. These are referred to as just-in-time (JIT) compilers or dynamic optimizers. Dynamic compilers offer the opportunity for run-time customization of applications to not only processor features that were unknown to the static compiler (e.g., those that are only present in a subset of a family of processors), but more importantly to dynamic events such as environmental conditions, load on the system, or application behavior. Parallelism can be throttled, remapped, or reorganized to take advantage of specific execution circumstances known only at run time.

STATIC COMPILATION

IDENTIFICATION AND EXTRACTION

Automatically parallelizing code poses two basic problems: finding an exploitable region of code through analysis and transforming the code into a parallel form. Analyses and transformations must work together to find and then exploit a specific family of parallelization. Finding code that cannot be transformed and transforming code that cannot be found is unhelpful.

MEMORY DEPENDENCE ANALYSIS

Data dependencies can greatly degrade the performance of parallelized code. In multicore architectures, cross-thread dependencies require communication or synchronization to maintain correctness. Synchronization and communication are undesirable since they are relatively expensive operations and force faster threads to wait on slower threads. By careful scheduling, an optimizing compiler can minimize the effect of data dependencies. Unfortunately, determining whether or not data dependencies exist between two instructions is nontrivial in general. This problem gave rise to a proliferation of memory dependence analyses that approximate the problem by indicating either that instructions MAY or MAY NOT alias. The quality of memory

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

dependence analysis strongly influences the performance of parallelized code.

Loops iterating over arrays dominate the run time of many signal processing applications. If the loop's iterations are independent, it can be explicitly parallelized on a multicore architecture or vectorized on a SIMD architecture. Array dependence analysis conservatively enumerates array dependences between loop iterations. Even when the iterations are not completely independent, regular patterns of array dependences can still find exploitable patterns. For instance, if all the elements in a row of a multidimensional array are dependent, but all the rows are independent of each other, then threads can execute each row in parallel. To exploit this type of opportunity, an optimizing compiler may transform a row-column loop into a column-row loop. The technique easily generalizes to higher dimensional arrays and iterating over a series of parallel diagonals.

[RESEARCHERS ARE NOW FOCUSING ON PROVIDING RUN-TIME FLEXIBILITY AND ADAPTATION FOR MULTICORE SYSTEMS.]

Array dependence analysis operates by encoding the constraints of the offset, or offsets in the case of multidimensional arrays, into a linear constraint problem. When the linear constraint problem admits no integer solution, there will be no aliasing between array elements between loop iterations. Unfortunately, determining that no integer solutions exist for a linear constraint problem is NP-hard. Therefore, compilers use heuristics. Pugh's Omega test [10] implements a complete algorithm with reasonable run time on real programs.

Pointer analysis attempts to determine whether two pointers can alias. In general, pointer analysis algorithms propagate facts about pointers throughout a program mostly using data flow analyses. The specifics of which facts propagate and how they propagate varies substantially among pointer analyses. For example, some analyses keep an explicit can-point-to set, others do not. Some analyses propagate information from a possible call site to all possible return sites, others match call and return sites accurately.

Shape analysis techniques use separation logic to determine the properties of recursive data structures (such as linked lists) directly from code. A shape analysis, for example, may conclude that no memory address recursively reachable from a tree's left child is recursively reachable from the tree's right child. Many of these analysis have been implemented in research compilers such as OpenImpact.

AUTOMATIC VECTORIZATION

Vector hardware performs the same operation on many inputs simultaneously. Analyses detect vectorization opportunities through array dependence analysis. If the dependence distance (the minimal distance between two dependent iterations) is less than the size of the hardware-specific vector, then the code can be vectorized.

The Cray-1 supercomputer, first introduced in 1976, featured Cray Fortran, the first automatic vectorizing compiler [11]. In its initial implementation, Cray Fortran could automatically vectorize Fortran DO loops without cross-thread dependencies, as determined by array dependence analysis, and without control-flow statements.

Thirty years later, IBM's state-of-the-art XL compiler for the Cell architecture relies on modern vectorization techniques to exploit the Cell's synergistic processing elements [7]. XL's vectorization is able to target a much broader collection of loops including loops with induction ("counting" in a regular way), deduction (computed by commutative operations such as sum, min, max, and product), loop-scoped, and loop-invariant accesses. Additionally, the XL compiler will also search individual basic blocks for vectorization opportunities. This is useful when a loop has already been completely unrolled. The XL compiler also implements loop distribution, allowing it to break a loop into several pieces each of which can be parallelized according to a different strategy. The advent of predicated vector operations allows vectorization of loops with control flow.

DOALL AND DOACROSS PARALLELIZATION

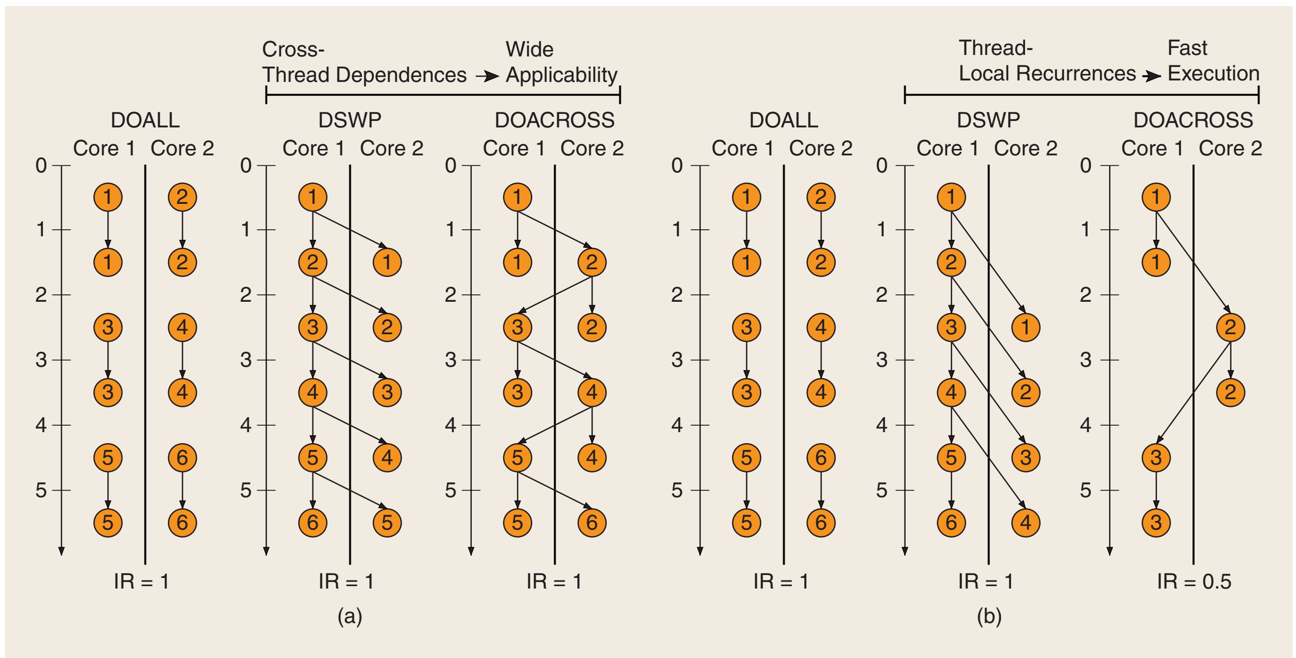
DOALL parallelization methods are the threaded multicore equivalent of vectorization techniques. If a loop has no cross-iteration dependencies and its iteration space is statically divisible, then the loop can execute in several threads in parallel with each thread covering a disjoint subset of the iteration space. DOALL parallelization methods are highly efficient since there is no cross-thread communication while the loop is iterating. However, the requirement that the iteration space be divisible prohibits pointer-chasing loops, which are common in general purpose applications. There have been successful attempts of DOALL parallelization in the research community such as the SUIF compiler system.

In DOACROSS parallelization, adjacent iterations execute in alternating threads. As soon as one thread has completed the loop's critical path, execution of the loop's next iteration begins on the next core while the current iteration is still being executed. In contrast to the restrictions imposed on DOALL loops, DOACROSS loops are universally applicable. However, DOACROSS loops only offer performance advantages when the critical path of the loop plus the time necessary to complete the critical path and to communicate between adjacent iterations is low relative to the amount of time spent executing code outside the critical path.

DECOUPLED SOFTWARE PIPELINING

Decoupled software pipelining (DSWP) partitions each loop iteration into a pipeline such that communication happens strictly from earlier stages in the pipeline to later stages [12]. Communication between stages is buffered, so unlike DOACROSS techniques, it is very latency tolerant. Figure 2 contrasts the effects of latency on the performance of DOALL, DOACROSS, and DSWP loops. Doubling the latency does not affect DOALL's performance, since DOALL loops have no

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.



[FIG2] Execution schedules of loops using DOALL, DSWP, and DOACROSS. Solid lines represent intra-iteration, and dashed lines represent loop-carried (critical path) dependences. Initiation rate (IR) is the number of iterations started per cycle. (a) Communication latency = 1. (b) Communication latency = 2.

cross-thread communication. Although DSWP engages in cross-thread communication, the communication is not on the critical path, so increasing latency increases run time by only a constant factor. For DOACROSS, communication from a previous iteration must complete before the next iteration can begin; increasing latency drastically diminishes DOACROSS's performance. The requirement that later stages in the pipeline not communicate with earlier stages, makes DSWP sensitive to the quality of memory analysis.

Figure 3 is an example of a DSWP parallelization. Figure 3(b) shows the dependencies in the original program [Figure 3(a)]. Read and print depend on prior invocations of themselves, but work is independent from prior iterations. The loop can be parallelized for four threads, with the first thread executing the read function and communicating the results to the second and third threads, which execute the work function of alternating iterations in parallel. Finally, the second and third threads communicate to the fourth thread which executes the print function. By reading from the second and third threads in alternating iterations, the fourth thread will maintain the original order of printing.

THE MAIN CHALLENGES THAT ARISE WHEN DESIGNING A RUN-TIME MANAGER FOR AN EMBEDDED SYSTEM IS THE MEMORY AND COMPUTATION OVERHEAD OF THE SYSTEM ITSELF.

STATIC MAPPING

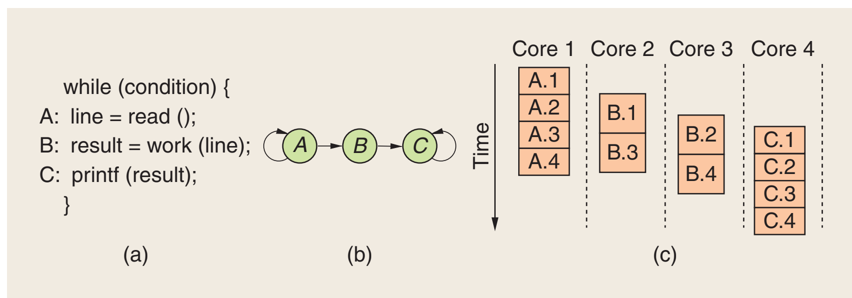
For pipelined parallelization, excessively long or unbalanced stages are undesirable, since the rate of execution is limited to at most that of the slowest stage. Excessively short stages are equally undesirable when the cost of communication between stages dominates the cost of the stages themselves. Therefore, an ideal static mapping from code to stages produces exactly enough stages to keep all cores busy, while minimizing communication and the latency of the slowest stage. DSWP operates on loop nests and attempts to balance the execution time of each stage using a simple greedy algorithm.

Stream graph modulo scheduling (SGMS) [13] is part of the StreamRoller compiler, which is a fully automatic compilation system that maps StreamIt application onto Cell architecture platforms. SGMS applies the traditional instruction-level modulo scheduling algorithm on a coarse-grain stream graph to pipeline the filters across multiple cores. The objective is to maximize concurrent execution of filters while hiding communication overhead to minimize stalls. This approach consists of two steps. First, an integrated filter fission and partitioning step is performed to assign filters to each processor to ensure maximum work balance. Parallel data filters are selectively replicated and split to increase the opportunities for evenly distributed work. The second step is stage assignment wherein each filter is assigned to a pipeline stage for execution. Stages are assigned to ensure data dependences are satisfied and interprocessor communication latency is maximally overlapped with computation. The result is a fully orchestrated stream program that resembles the right hand portion of Figure 1(d).

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

MEMORY AND COMMUNICATIONS OPTIMIZATIONS

While identifying and managing parallel computation is important for multicore systems, inefficient use of the memory system can cause cache stalls to dominate program execution. For array-dominated applications, compiler transformations can overcome this problem. Loop tiling or loop blocking is a technique where a compiler will reorder the iterations of a loop to achieve better memory locality and thus better caching behavior. For instance, if a loop updates in row-column order and an entire row cannot fit in the cache at once, then the data loaded into the cache gets evicted before being reused. Instead, loop tiling will break the iteration space into a series of small enough squares to fit in the cache. The cache hit rate and thus performance will increase.



[FIG3] Example compiler parallelization using decoupled software pipelining to a traditionally sequential while loop. (a) Example code. (b) Static stage dependences. (c) Potential task execution.

RUN-TIME MANAGEMENT AND OPTIMIZATION

Exposing the parallelism present on the underlying hardware can be a double-edged sword. While compilers and programmers can leverage their knowledge about the underlying hardware to perform various optimizations, the resulting code may then become tightly tied to the underlying system, limiting the true "portability" of the program. Correct execution may be possible on various hardware configurations, but the program is likely optimized for just one of those configurations. Even when executing on the specific, targeted configuration, performance can suffer when unexpected run-time events (such as cache misses) or resource contention occur. A lack of flexibility to adapt to unexpected events can be a significant downfall and is in fact considered to be a major weakness of statically scheduled VLIW codes. Therefore, researchers are now focusing on providing run-time flexibility and adaptation for multicore systems.

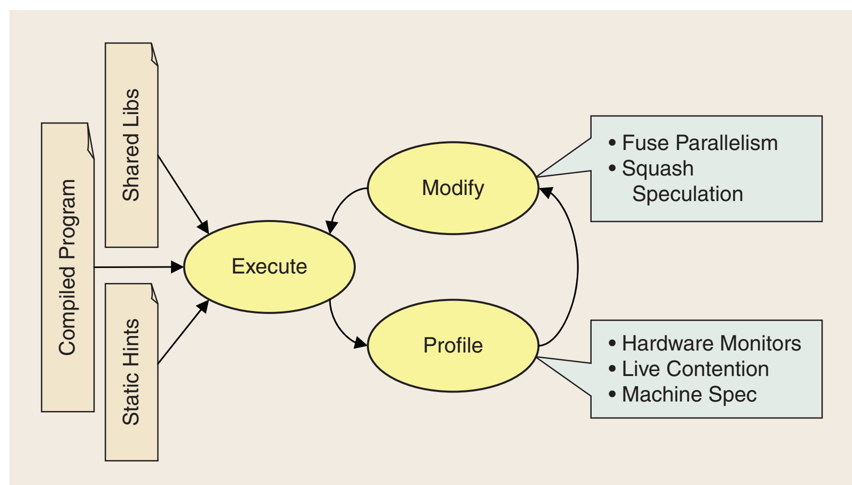
Aside from adapting to specific hardware resources, several run-time events present additional opportunities for dynamic adaptation and optimization. For one, system utilization changes (from competing tasks) at run time will mean that the number and type of available computational resources is in a constant state of flux. Meanwhile, the behavior of an application itself changes over time as it moves through various application phases. Phased behavior has been widely investigated in the high-performance computing arena and has been shown to present numerous opportunities and challenges. Above all, phase changes should result in periodic re-evaluation of compile-time decisions.

[THE DIFFICULTY AND COMPLEXITY OF PROGRAMMING FOR MULTICORES INEVITABLY PUSHES PROGRAMMERS TO RELY MORE HEAVILY ON TOOLS, SUCH AS COMPILERS AND RUN-TIME OPTIMIZERS.]

Finally, both transient and persistent symptoms may arise that relate to the temperature and reliability of the system. For instance, it is possible to a) adjust a sequence of instructions to change the temperature profile (often at the expense of performance), b) adjust computation to avoid unreliable hardware, or c) adjust the fidelity of certain applications in response to depleting battery power.

RUN-TIME SUPPORT

As depicted in Figure 4, an application can be regularly profiled and adjusted at run time. Dynamic adaptation can be enabled by a variety of means, including compile-time multiversioning, JIT compilation with continuous optimization, or dynamic binary translation. Each of these mechanisms allows an application to be tailored to a particular



[FIG4] A run-time adaptation engine continuously profiles and modifies a program as it runs. Profile information can come from the hardware, the operating system, the compiler (via statically inserted hints), or the application itself.

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

hardware platform and run-time environment.

Compile-time multiversioning involves embedding several versions of a particular function into a compiled binary, and using a dynamic trigger to select the appropriate version at run time. The version that is ultimately executed can remain fixed for the duration of a program's execution time, or it can vary, based on some dynamic factor.

JIT compilation involves postponing the act of machine code generation until the application executes. A static compiler generates an intermediate representation that targets a generic, stack-based architecture. Then, at run time, a machine specific compiler converts the intermediate code into native machine code. Adaptive JIT compilers will then regularly revisit the generated code to determine whether to apply additional optimizations to the code.

Finally, dynamic binary translation systems are software-based systems that take previously compiled machine code and modify it at run time. These systems have the benefit of functioning on legacy code that was written in any language and was compiled with any compiler.

ONLINE APPLICATION MONITORING AND PROFILING

The first step in supporting run time adaptation is to provide and leverage support for application profiling and system monitoring. A run-time system can gather information about an executing application for a variety of reasons. An obvious reason is to uncover bottlenecks, which can be reported back to the programmer. Another reason is to gather information that can be used to trigger run-time adaptation.

The profile information used to trigger program transformations is less likely to be aggregated over an entire run but is instead a means for efficiently detecting critical anomalies. The information can come in a variety of forms:

1) Run-time program inputs will often trigger comparisons with expected values and assumptions that were made at compile time. Any significant deviation from these expectations can trigger transformations.

2) Run-time performance and resource contention, as gathered by hardware performance counters, will also adjust the level of parallelism attempted at run time.

3) System load and competition from colocated threads, as measured by the operating system, can be used to trigger a variety of rescheduling and remapping events.

4) Temperature and reliability anomalies can trigger code reoptimization and resource avoidance, and battery information can trigger algorithm fidelity adjustments.

DYNAMIC ADAPTATION

Given the appropriate profile trigger, various opportunities exist for dynamically transforming programs to adapt to changing conditions. Options include a) removing excessive parallelism that the run-time environment cannot support, either due to hardware limitations or due to contention from other processes, b) verifying the correctness of compile-time assumptions and modifying the application accordingly, c) leveraging run-time information to detect additional opportunities for parallelism, and d) rescheduling certain application threads to execute on processor cores with less contention, lower temperatures, higher reliability, and/or more synergy with colocated threads.

In summary, exploiting parallelism requires a multi-pronged approach. A static compiler can perform the task of aggressively maximizing the potential for parallelization assuming unbounded resources and no system load. It will be the task of the run-time system to then adapt the parallelism to the available hardware resources as they change over time, and also to exploit further parallelization opportunities based on dynamic behavior.

CONSIDERATIONS FOR SIGNAL PROCESSING

While run-time adaptation is important for managing parallelism in the high-performance computing domain, adaptation is critical for signal processing applications for two reasons: a) the highly dynamic nature of the applications and operating environments and b) the heterogeneity of the devices. Each of these traits presents challenges to the DSP software developer (who must write device-specific code for each and every platform) and to the compiler (which must attempt to predict and optimize for a variety of dynamic events).

The main challenges that arise when designing a run-time manager for an embedded system is the memory and computation overhead of the system itself. JIT compilers, multiversioned code, and dynamic binary instrumentation systems all consume memory and require additional computational resources. This overhead must be offset by the adaptation, or alternatively, the benefits to the programmer must be significant enough for the overheads to be tolerable. Researchers have already made great strides toward reducing the footprint and overhead of adaptation engines. Meanwhile, the benefits to the DSP software developer for masking the heterogeneity of the underlying system are long overdue.

FUTURE TRENDS

As multicore becomes the de facto platform in computing, signal processing algorithm and software developers face new and difficult challenges. First and foremost, parallelism rather than sequential measures of work (e.g., instruction count) become the critical factor for performance. Coarse-grain parallelism must be exploited to extract meaningful performance gains by spreading out work across cores. We expect data, loop, and pipeline parallelism to dominate the landscape, since they are

[AS MULTICORE BECOMES THE DE FACTO PLATFORM IN COMPUTING, SIGNAL PROCESSING ALGORITHM AND SOFTWARE DEVELOPERS FACE NEW AND DIFFICULT CHALLENGES.]

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.

easiest to identify in media and signal processing applications. However, with this challenge comes the opportunity of harnessing enormous single-chip computing capabilities. Platforms such as the NVIDIA GeForce GTX 295 provide peak performances of over 1.7 teraflops at a modest cost, while platforms such as ARM Ardbeg provide tens of giga-operations/s in less than 300 mW.

The second challenge is that the complexity of signal processing algorithms is expected to continue growing. Complexity ranging from more signal formats and processing standards to more complex encoding methods and usage scenarios will force developers to produce more sophisticated software implementations. More complex code will require traditional assembly and low-level C implementations to utilize better software engineering methodologies and libraries provided by object oriented programming (e.g., C++ and Java). As a result, signal processing applications will start to resemble general-purpose programs as opposed to traditional scientific applications. Further, applications will have less statically predictable behavior, requiring runtime systems to continually adapt to the changing performance demands of the application. Harnessing and managing this complexity without the aid of software development tools will quickly become a skill that only few programmers possess. Thus, the signal processing community must embrace compilers and other software development tools to manage this complexity and develop effective applications for multicore systems.

AUTHORS

Mojtaba Mehrara (mehrara@umich.edu) received the B.S. degree in electrical engineering from Sharif University of Technology, Iran, in 2005 and the M.S. degree in computer science and engineering from the University of Michigan in 2007. He is a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. His research interests include compiler and architecture techniques for improving the performance and programmability of parallel systems.

Thomas Jablin (tjablin@cs.princeton.edu) received the B.A. degree from Amherst College in 2006 and the M.A. degree in 2009 from Princeton University. He is a P.hD. candidate in the Department of Computer Science at Princeton University. His research interests include compiler-driven automatic parallelization and dynamic compilation. He is a student member of the ACM.

Dan Upton (upton@virginia.edu) received a B.S. degree in computer science from the University of Richmond in 2005. He received his master's degree in computer science in 2008 and is currently pursuing a Ph.D. at the University of Virginia. His research interests include thermally aware computing and efficient full-system profiling and analysis.

David August (august@princeton.edu) is an associate professor in the Department of Computer Science at Princeton University, where he directs the Liberty Research Group. He earned his Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign, where he worked as a member of the IMPACT research compiler group. His work has been recognized by numerous best paper awards including two IEEE Micro "Top Picks."

Kim Hazelwood (hazelwood@virginia.edu) is an assistant professor at the University of Virginia. Her research focuses on virtualization and run-time adaptation. She received her Ph.D. degree in computer science from Harvard University in 2004, followed by a postdoc at Intel, where she helped to develop the Pin dynamic instrumentation system. She has received the FEST Young Investigator Award, an NSF CAREER award, a Woodrow Wilson Career Enhancement Fellowship, and the Borg Early Career Award.

Scott Mahlke (mahlke@umich.edu) is an associate professor in the Electrical Engineering and Computer Science Department at the University of Michigan, where he leads the Compilers Creating Custom Processors Group. He received the Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1997. He was named the Morris Wellman Assistant Professor in 2004 and won the 2007 Most Influential Paper Award from the International Symposium on Computer Architecture.

REFERENCES

[1] J. H. Ahn, M. Erez, and W. J. Dall, "Tradeoff between data-, instruction-, and thread-level parallelism in stream processors," in *Proc. ICS'07*, 2007, pp. 126–137.

[2] J. A. Fisher, P. Farabosch, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compiler and Tools*. San Mateo, CA: Morgan Kaufmann, 2004.

[3] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proc. Microprocessor Forum*, Oct. 2007, pp. 103–104.

[4] T. D. R. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon, "Biomedical image analysis on a cooperative cluster of GPUs and multicores," in *Proc. 2008 Int. Conf. Supercomputing*, pp. 15–25.

[5] A. Ruiz, M. Ujaldon, L. Cooper, and K. Huang, "Non-rigid registration for large sets of microscopic images on graphics processors," *J. Signal Process. Syst.*, vol. 55, no. 1–3, pp. 229–250, Apr. 2008.

[6] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P.Hanrahan, "Larrabee: A many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 29, no. 1, pp. 10–21, Jan./Feb. 2009.

[7] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind, "Optimizing compiler for the CELL processor," in *Proc. 14th Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2005, pp. 161–172.

[8] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *Proc. 14th Int. Symp. High-Performance Computer Architecture*, Feb. 2008, pp. 290–301.

[9] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. 2002 Int. Conf. Compiler Construction*, 2002, pp. 179–196.

[10] W. Pugh, "The Omega test: A fast and practical integer programming algorithm for dependence analysis," in *Proc. 1991 ACM/IEEE Conf. Supercomputing*, 1991, pp. 4–13.

[11] R. M. Russell, "The CRAY-1 computer system," *Commun. ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[12] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proc. 38th IEEE/ACM Int. Symp. Microarchitecture*, Nov. 2005, pp. 105–116.

[13] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proc. SIGPLAN '08 Conf. Programming Language Design and Implementation*, June 2008, pp. 114–124.

Authorized licensed use limited to: University of Michigan Library. Downloaded on October 26, 2009 at 14:54 from IEEE Xplore. Restrictions apply.