Hardware parallelism vs. software parallelism

John A. Chandy Janardhan Singaraju
Department of Electrical & Computer Engineering
University of Connecticut

Abstract

In this paper, we explore the rationale for multicore parallelism and instead argue that a better use of transistors is to use reconfigurable hardware cores. The difficulty in achieving software parallelism means that new ways of exploiting the silicon real estate need to be explored. Hardware implementations can often expose much finer grained parallelism than possible with software implementations. We discuss some of the challenges from a design and system support perspective.

1 Background

In recent years, microprocessor design has hit a clock cycle wall as energy consumption has limited the clock scaling that was the norm for three decades. Since transistor scaling has continued unabated, the design path of choice has been to use these transistors to add multiple cores. Intel's commodity microprocessors currently have up to 6 cores and within the next year 8 core processors are expected. Intel has already demonstrated 80 cores in a 45 nm process [1]. Startups such as Tilera are shipping 64 core processors at 90nm. Graphic processors have been even more aggressive as evidenced by NVidia's recent GeForce 280 GPU with 240 stream processors on a 65 nm process. Within five years fabrication process technology is expected to reach 11 nm, and as a result, it is not unreasonable to expect within that timeframe over 100 cores in commodity microprocessors and over 1000 cores in GPUs and specialty microprocessors.

The difficulty with this future road map is how end users are supposed to use these processors. It is clear that there are data intensive and compute intensive parallelizable applications that can use these parallel processing capability. One only needs to see that the largest supercomputers in the world are running applications such as large-scale simulation, genome sequencing, and data mining - applications that would easily map to these many core CPUs. However, supercomputer applications are not enough to drive commodity processors. Thus, the challenge is to find consumer and mainstream applications that can take advantage of the abundance of cores.

In current multicore CPUs, the primary usage of the extra cores has been to support multitasking - i.e. the many processes that run in the background such as virus checking, indexing, defragmentation, etc. However, multitasking is unlikely to scale past about 8 cores. The challenge is to develop applications that can go beyond multitasking and use parallelism to utilize the cores. Decades of research have shown that parallelism is difficult to find in typical applications whether by hand coding or automated compiler techniques. The only applications that easily scale to 100's or 1000's of cores or those that can be decomposed into independent tasks or those that operate on independent sets of data. Examples of such mainstream applications include image editing, rendering, and search. However, overall, the set of applications that are easily parallelizable is limited. Therefore, going past 8-16 cores is unlikely to provide benefits to most application workloads.

One option to adequately use the silicon area is hybrid multicore architectures. For example, gaming systems have been huge beneficiaries of the 100+ core capabilities of high end GPUs. That realization has led to the proposal of heterogeneous multicores where streaming graphic processors are integrated with traditional CPUs. The Cell Broadband Engine is the best known example of such an architecture where a POWER CPU is augmented with 8 Synergistic Processing Elements that can handle various types of data streaming operations [2].

With the potential of thousands of available cores, the question becomes what cores should we choose on a heterogeneous processor. Since the amount of readily accessible parallelism in commodity applications is limited, 8-16 CPU cores is probably sufficient to handle multitasking and small-scale parallelism. 200+ GPU cores can handle graphics and multimedia data streams. A few DSP cores may be useful to handle communications. What do we do with the remaining cores?

2 Reconfigurable Hybrid Multicore Architecture

Our belief is that finding 100-way parallelism in mainstream software is a lost cause, and instead the place to look for parallelism is in hardware. In other words, the

Algorithm	Speedup	FPGA	CPU
DES Encryption [3]	24	Garp 133 MHz	SPARC 167 MHz
Number Factoring [4]	6.8	Xilinx XC4085 16 MHz	UltraSPARC 200 MHz
Intrusion Detection [5]	27.8	Xilinx Virtex2 303 MHz	Pentium 4 1.7 GHz
Numerical Simulation [6]	5.69	Xilinx Virtex4 50 MHz	Intel P4 3.0Ghz
Genome Sequencing [7]	100	Xilinx Virtex4 125 MHz	AMD Opteron 2.2 GHz

Table 1: Hardware to software speedup

remaining cores should be used to provide hardware that can be configured to implement a wide variety of logic functions - a reconfigurable fabric as found in current FPGAs. The rationale is that reconfigurable hardware is a better use of transistors than processor cores that may not be fully utilized because of the difficulty in finding adequate parallelism. Reconfigurable hardware can often find more parallelism in an algorithm because there are no context switching costs allowing blocks to be at a much finer granularity. In addition, bitwise parallelism is much more readily available than task parallelism.

This has been recognized in work by Williams et al where the *computational density (CD)* of various architectures is measured [8]. CD is a metric that represents the computational capability available in a silicon die. Williams et al found that reconfigurable devices have the best computational density for bit-level and integer operations while traditional CPUs do best for floating point operations. This result would seem to indicate that the best utilization of resources on a many-core die is to make some of those cores reconfigurable to support integer operations rather than make more CPU cores that are best suited for FP operations that are rarely used in general purpose applications. Table 1 shows speedups for various algorithms on an FPGA compared to software algorithms. Hardware provides significant speedups at significantly lower clock speeds. Sirowy and Forin showed that hardware implementations can be more efficient by eliminating fetch overhead, enhancing instruction parallelism and more pipelining [9].

The architecture that we are envisioning would have CPU cores, stream processing cores (SPC), reconfigurable hardware, and multi-ported memory to supports access from multiple cores. We call this a *Reconfigurable Hybrid Multicore Architecture (RHyMA)*. We assume some network on chip such as a mesh is available. Other implementations of a RHyMA may include different types of cores including vector units, DSPs, peripherals, and other specialized functional units.

The merging of computational cores with reconfigurable cores is not new as there were several research reconfigurable computing architectures proposed in the 1990s [10, 11, 12]. However, these architectures did not enter into the mainstream for two reasons: chip real estate was better used for computation at that time and hardware design is difficult. We believe the first reason is no longer an issue since hardware resources are now more abundant. The second reason is still problematic but we present a design framework that hopefully addresses this issue somewhat.

3 RHyMA Software and OS Support

In order to facilitate this transformation to RHyMA, a key requirement is software support for such architectures, particularly in terms of tools to develop applications that can run on reconfigurable multicore processors.

3.1 Design Flow

While we have made the case that microprocessor architects must begin to adopt reconfigurable cores within their processors, RHyMA will only be viable with adequate general software support. This includes both operating system support as well as enhanced design tools. Operating systems must support multithreaded and multicore environments as well as capabilities to load and unload configurations, real-time repartitioning, and allocation of reconfigurable hardware resources.

Recent efforts such as OpenFPGA [13] and PFIF [14] provide standardized software APIs to hardware IP cores. These APIs provide an interface between software and hardware thereby allowing software to call hardware functions by specifying mechanisms to pass data to and from the hardware. These APIs allow software to work with hardware and allow hardware designers to develop portable IP core libraries. However, there are no mechanisms to manage the loading and unloading of cores, include software implementations or support multiple hardware implementations. Therefore, the operating system must support mechanisms for dynamically loading these libraries as well as possibly choosing between multiple implementations of various cores.

We have currently developed a design flow for RHyMA application development as shown in Figure 1. The design flow has two paths, one for software and one

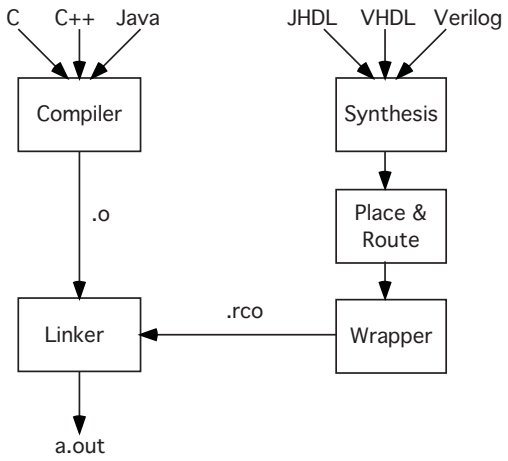

Figure 1: *Hybrid Computing Design Flow.*

for hardware. The software path can use standard software design languages like C, C++, Java, Fortran, etc. Tasks that can be implemented in hardware will be implemented in appropriate specification languages such as VHDL or Verilog. Note that even though a particular task may be implemented in hardware, there should still be a software implementation available. This software implementation is executed when the hardware implementation on the array is not available, for example during reconfiguration or if array space is not available.

Since the software design path can use standard compilers, the hardware design path is the more interesting of the two. The expectation is that experienced hardware designers would make a set of libraries available for software developers to use in applications. The design process has three main components: synthesis, place and route (P&R), and wrapper generation. The synthesis process takes a hardware description specification and transforms it into a circuit netlist. The P&R process outputs a bitstream which describes the configuration of the reconfigurable array. When a task is instantiated, the amount and shape of the space available on the array will vary and thus require different placements at runtime. We can create bitstreams at compile time for multiple static configurations for a particular hardware task. These multiple configurations would exist in the object file for use at runtime as described in more detail in Section 3.2. Since partial reconfiguration is limited to a fixed number of possible orientations, the number of configurations that would need to be generated and stored is small.

3.2 Multiple Hardware Implementations

Besides reverting to the software implementation, the design flow also provides another fallback. It is possible to specify multiple hardware implementations of the same task with varying levels of performance area tradeoffs. For example, one high performance implementation may use several degrees of hardware parallelism and a second more compact implementation may not use any parallelism at all. In systems where power consumption is a concern, the design may also include energy friendly implementations that can be instantiated if necessary.

It is the responsibility of the wrapper tool to coalesce these multiple implementations of a hardware task and generate a .rco or reconfigurable computing object file. The wrapper tool takes in a C++ or Java API file that describes the various implementations and their interfaces and behaviors. A sample API wrapper file may look as shown below. The API is similar to the JHDL API to instantiate configurations. The difference is that the API wrapper that we propose is an interface to the operating system to let it manage configurations rather than have the application manage configurations directly.

```
class encrypt_rc : rc_wrapper {
public:
  void unload_state() { ... }
  void load_state() { ... }
  void execute( ... ) { ... }
  void wait( ... ) { ... }
  virutal void repartition( ... ) { ... }
}

encrypt_rc::encrypt_rc()
{
  register_implementation( slow_encrypt_entity );
  register_implementation( fast_encrypt_entity );
  register_sw_implementation( sw_encrypt );
}
```

Each hardware task requires an API file where the task is defined as a subclass of the rc_wrapper class. The required methods are `unload_state` which unloads state from the hardware to memory, `load_state` which loads the state back into the hardware, and `execute` which is called to initiate the hardware task. `unload_state` and `load_state` are called on context switches. `execute` will transfer parameter data to the hardware block and start the execution asynchronously and `wait` waits for the hardware to complete. The optional `repartition` method will be called to do repartitioning of the task. Typically, this method would shrink the implementation in a prescribed fashion rather than requiring multiple hardware implementations. For example, a hardware cache could be reduced in size programmatically. The constructor for the class registers the available software and hardware implementations with the run-time system. The wrapper tool compiles the API file and then copies the bitstreams from the P&R process into the data section of the `.o` file. The resulting output file is the `.rco` file.

The final step in the design flow is the linker, which creates the executable. The RHyMA linker performs the same functions as a normal linker by resolving unbound variables and names. However, two main differences are apparent. First, function calls that resolve to functions having hardware and software implementations will be replaced by a call to the wrapper API's `execute` call. This allows OS management of the hardware task execution. The second difference is a modification of the executable so that the OS can be made aware of any hardware tasks that may be called during the execution of the file. We have successfully created the wrapper functionality for FPGAs with embedded CPUs and are currently extending it to support partial reconfiguration.

3.3 Task Execution

With RHyMA one may have the luxury to dedicate reconfigurable hardware to a particular task. It is more likely that, you will find multiple threads running on the processor, thus requiring the ability to reconfigure the hardware as threads change. Thus, mechanisms are necessary to accommodate two threads sharing a reconfigurable hardware. This will require support for real-time partitioning of the reconfigurable hardware. As new threads enter the system, repartitioning may be required as well as support for unloading and loading application kernels. As kernels get removed from the reconfigurable hardware, this will require architectural support for state retention across context switches.

A RHyMA-aware OS loads executables into the RHyMA array, begins the execution, and then manages the scheduling of the task. As an example, consider a web browser that uses SSL encryption. Because the browser has been dynamically linked to an RHyMA-aware SSL library, the browser can automatically take advantage of a hardware SSL implementation. When the OS loads the browser for execution, the `a.out` header tells the OS that the process will use the dynamic SSL library which in turn tells the OS that the SSL library has potential hardware implementations. The OS will scan the possible hardware implementations and load them onto the reconfigurable hardware. The configuration of the hardware can take place in parallel with the execution of the browser. This is important, in that process startup is not blocked waiting for loading of the hardware particularly when it is possible that the hardware may never get used (for example, the user never visits an encrypted page). This new configuration is loaded into a shadow context again in parallel with CPU execution as well as main context execution. The first time the task is called, the OS determines if the configuration load has completed. If not, the OS runs the software implementation. If the load has completed, then the OS calls the task's `execute` function to complete the task.

If the RHyMA array is already being used by an existing process, the OS must initiate a partitioning and placement routine which will allow the array to be shared. The OS must also decide whether to 1) keep the existing configuration and let the new function run in software or 2) remove the existing configuration and let the new function run in hardware thereby consigning the old function back to software or 3) use smaller configurations of both functions. This question must be answered with several metrics in mind including the performance, priority and CPU load of each thread.

Multicore execution adds a different dimension to the problem since tasks on different CPU cores are possibly sharing reconfigurable logic. If the CPU cores are using the same hardware cores, the choice is whether to partition the reconfigurable array amongst multiple cores with smaller and lower performance configurations or to use a full hardware configuration and then block cores from accessing the hardware until the computation is complete. Depending on the length of execution of the computation, blocking may be more desirable.

4 Related Work

The integration of reconfigurable logic with CPUs is a well known idea and reconfigurable computing has been an active research area for many years. The primary characteristic of reconfigurable computing is the integration of a microprocessor with programmable hardware. Our work builds on this existing work and offers a design platform required to make RHyMA or any of these earlier architectures a reality. There have been several reconfigurable computing architectures proposed by the research community and they fall into two general classes - functional unit based and coprocessor based.

Functional unit based reconfigurable computing takes a microprocessor and integrates the reconfigurable hardware as a functional unit within the microprocessor. Examples of these architectures include Chimaera [10], PRISC [11], and OneChip [12]. These reconfigurable functional units (RFU) execute custom instructions to provide speedups of short instruction sequences. The advantage of the RFU design is that the tight coupling with the processor core allows fast access to processor registers. Intelligent compilers are used to identify blocks of code that can be mapped to a RFU [15].

With coprocessor based designs, the reconfigurable hardware is distinct from the main processor core, in that it does not participate in the pipeline. How-

ever, it may use some processor functionality such as memory access or data caching. Examples of coprocessor based research reconfigurable platforms include Garp [16] PipeRench [17], DISC [18], and PRISM [19]. Coprocessor architectures have become common in commercial high performance computing systems such as the Cray XD1 and SGI RC100. In a variation of the coprocessor architecture, Xilinx and Altera have both introduced FPGAs with embedded processor cores. The RHyMA architecture that we have presented is a form of coprocessor based design.

5 Conclusions

We have argued for multicore parallelism in hardware instead of software and presented some of the challenges in making a reconfigurable hybrid multicore architecture viable particularly in terms of design tools and operating system support. Much of the issues are due to software management particularly with respect to software design, multitasking and library support. We have proposed some strategies for hardware-software cosynthesis and linking and execution. Operating system support requires support for task management, dynamic RC libraries, and power management.

6 Acknowledgment

This work was supported in part by the National Science Foundation High End Computing University Research Activity program under award number CCF-0621448.

References

[1] J. Held, J. Bautista, and S. Koehl, *From a Few Cores to Many: A Tera-scale Computing Research Overview*, Intel Corporation, 2006.

[2] M. Gshwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," *IEEE Micro*, pp. 10–24, March-April 2006.

[3] J. R. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," in *Proceedings of the IEEE Symposium on FPGA-Based Custom Computing Machines*, Apr. 1997, pp. 12–21.

[4] H. J. Kim and W. H. Mangione-Smith, "Factoring large numbers with programmable hardware," in *Proceedings of the ACM/SIGDA Symposium on FPGAs*, 2000, pp. 41–48.

[5] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for efficient and high-speed NIDS pattern matching," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2004, pp. 258–267.

[6] C. He, W. Zhao, and M. Lu, "Time domain numerical simulation for transient wave equations on reconfigurable coprocessor platform," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 2005.

[7] O. O. Storaasli, "Accelerating genome sequencing 100x with FPGAs," in *High Performance Embedded Computing Workshop*, 2007.

[8] J. Williams, A. D. George, J. Richardson, K. Gosrani, and S. Suresh, "Computational density of fixed and reconfigurable multi-core devices for application acceleration," in *Proceedings of Reconfigurable Systems Summer Institute*, Urbana, IL, Jul. 2008.

[9] S. Sirowy and A. Forin, "Where's the beef? Why FPGAs are so fast," Microsoft Research, Tech. Rep. TR-2008-130, 2008.

[10] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable unit," in *Proceedings of the International Symposium on Computer Architecture*, Jun. 2000.

[11] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the International Symposium on Microarchitecture*, 1994, pp. 172–180.

[12] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proceedings of the IEEE Symposium on FPGA-Based Custom Computing Machines*, Apr. 1996.

[13] M. Wirthlin, D. Poznanovic, P. Sundararajan, A. Coppola, D. Pellerin, W. Najjar, R. Bruce, M. Babst, O. Pritchard, P. Palazzari, and G. Kuzmanov, "OpenFPGA CoreLib core library interoperability effort," *Parallel Computing*, vol. 34, pp. 231–244, May 2008.

[14] M. Huang, I. Gonzalez, S. Lopez-Buedo, and T. El-Ghazawi, "A framework to improve IP portability on reconfigurable computers," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Jun. 2008.

[15] Z. A. Ye, N. Shenoy, and P. Banerjee, "A C compiler for a processor with a reconfigurable functional unit," in *Proceedings of ACM/SIGDA Symposium on Field Programmable Gate Arrays*, Feb. 2000.

[16] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *IEEE Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[17] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A coprocessor for streaming multimedia acceleration," in *Proceedings of the International Symposium on Computer Architecture*, May 1999.

[18] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Apr. 1995, pp. 99–107.

[19] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, no. 3, pp. 11–18, Mar. 1993.