Hardware / Software Trade offs in Multicore Architectures

Steven A. Guccione
Cmpware, Inc.
Austin, TX (USA)
Steven.Guccione@cmpware.com

Abstract - For over three decades microprocessor architectures have used increasingly large numbers of transistors to improve performance. With the advent of multicore microprocessors, it is not obvious how best to use transistor budgets. The choice is between a small number of complex cores or a larger number of simpler cores. This paper investigates the specific trade offs between using a hardware floating point unit as opposed to software emulation of floating point using a larger number of cores.

I. Introduction

The history of microprocessor architecture has been one of making use of ever increasing numbers of transistors to provide ever increasing levels of performance. Over the course of several decades the number of transistors available to microprocessor architects could reliably be expected to double approximately every 18 months.

While the number of available transistors increased by several orders of magnitude over this period, the performance gains achieved were much smaller. New transistors added to maturing processor architectures achieved smaller and smaller incremental performance gains over time until finally large numbers of transistors did little to increase performance for most applications.

Near the beginning of the new millennium a significant change occurred in mainstream microprocessor architecture. New transistor budgets were no longer used to increase performance of individual microprocessors; instead multiple processor cores were implemented on a single device. This change occurred for a variety of reasons, but the effect on performance was dramatic. Suddenly doubling the number of transistors could, at least in theory, double the performance of the device.

This was a situation that had not existed for decades. While mainstream microprocessor architects quickly embraced this multicore approach to utilizing new transistor budgets, the process has typically made use of existing modern processor architectures for the individual cores. Since it is clear that the last several generations of architectural enhancements added little in performance to these single core architectures, it seems obvious that re-deploying these transistors to increase the number of cores, while reducing the size of the cores themselves, should result in an increase in overall performance. What is not clear is how far back should the clock be turned in redeploying transistors in this manner.

This paper will focus on a very early hardware addition to the basic microprocessor architecture: the floating point unit. Issues concerning the redeployment of floating point unit transistors to increase the number of cores in a multicore device is explored. Software emulation of floating point operation as well as similar techniques are examined.

II. Multicore Architectures

All modern desktop and server microprocessor devices today are multicore devices. It is expected that this trend will continue for some time, with the number of cores continuing to increase, approximately doubling with every new generation of microprocessor.

This shift to multiple core microprocessors has occurred for a variety of reasons, which are interrelated and happened to occur at approximately the same time in history. First, the increase in clock speed and the continued shrinking in transistors resulted in a situation where power consumption reached and even exceeded 100W per device. The trend in power consumption for high performance microprocessors was perhaps the most visible and urgent reason for the move to multicore. With multiple cores, the core voltages and clock speed could be reduced, resulting in significant power savings. Two lower power, but lower performing, cores could operate at a combined performance level above that of a more powerful single core.

A second reason for the move to multicore was the plateauing of microprocessor clock speeds. While related to power consumption, the ability of modern CPU architectures to support clock speed above approximately 4 GHz was also a function of the underlying silicon technology. In order to continue to provide performance gains in new generations of microprocessors, other techniques besides faster clocks were going to have to be employes. Multicore was a convenient method of boosting total processor device performance without having to raise clock speeds.

Finally, one significant problem with modern microprocessor design was the cost. Design and verification of a large microprocessor was beginning to exceed the available resources of even the largest corporations. Multicore also addressed this problem by reusing a single smaller core multiple times in the same design.

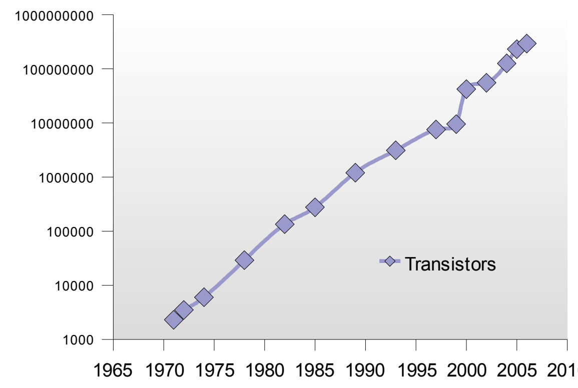

Figure 1: Transistor budgets for microprocessors.

As Figure 1 indicates the number of transistors available to microprocessor architects continues to grow, even as the ability to produce traditional single core microprocessor devices became less and less feasible. After approximately the year 2000, multicore became the sole method of continuing to keep microprocessor performance on the established 30 year track.

III. FPGA CPUs

It is clear that the later generations of microprocessor devices used their transistor budgets to provide fairly minimal increases in performance. In fact, much of the performance gains in modern microprocessor devices came from increases in clock speed associated with smaller transistors, not from the use of larger numbers of these smaller transistors. While these performance gains were important in the highly competitive marketplace, the equation changed significantly with the move to multicore.

While in early microprocessors, a doubling in the number of transistors could result in a doubling of performance, only very small gains could be expected in recent generations. But with mucticore promising a doubling in performance for a doubling in transistor count, a re-evaluation of existing multiprocessor architectures may be in order. Such an investigation is potentially complicated due to the long history, lack of published information and potential effects of rapidly changing technology.

Fortunately, Field Programmable Gate Array (FPGA) technology presents an environment remarkably similar to the world of microprocessor design circa 1985. Simple 32-bit RISC processors are beginning to find their way in common usage and hardware floating point units are just beginning to be explored.

Component	*LUTs*
CPU	800
FP Add	1,312
FP Multiply	1,380
FP Mac	2,772
FPU support*	850

Table 1: FPGA core sizes.

Table 1 gives a list of microprocessor core components for FPGAs measured in the number of 4-input Look Up Tables (LUTs) used in each component. LUTs are the basic building block used by FPGAs to construct circuits. This provides a very good measure of the relative circuit size of these elements.

While there is currently no information available on a complete floating point unit Arithmetic and Logical Unit (ALU) in an FPGA, these numbers indicate that a Multiply-Accumulate unit (MAC) is over three times the size of the CPU core itself. In a multicore environment, the choice of hardware for such a system would be a single CPU with a single MAC or approximately five CPUs. In fact, the choice is likely to involve a larger number of cores. The choice may be between 200 CPU plus MAC cores and 1,000 CPU cores.

IV. Floating Point Performance

With a hardware design choice of a single CPU plus a single FPU versus five CPUs, some measurement of performance of these competing approaches is required. In this example, the goal is to first give an estimate of hardware floating point performance versus software emulation of floating point. Such a system may offer a wide variety of size versus performance points. While floating point hardware often takes multiple cycles and has the ability to pipeline certain operations, it is conservatively assumed that each floating point operation is independent and takes place in a single cycle. These two approaches were simulated on the Cmpware CMP-DK multicore simulator [8]. The results of these simulations are shown in Table 2.

```
int main(int argc, char *argv[]) {
      int  i;
   float  a = 10.5;
   float  b = 3.25;
   float  c = 0.0;

   for (i=0; i<1000; i++)
      c = c + (a * b);

   }  /* end main() */
```

Figure 2: The multiply-accumulate code.

Table 2 indicates that unoptimized floating point software emulation is approximately 28 times slower than the floating point hardware.

While this result is disappointing, attempts to use the compiler to optimize the code results in a factor of two improvement in performance over the unoptimized floating point emulation. Unfortunately it also results in a factor of three increase in performance for the floating point hardware architecture. Overall, the optimized code for the software emulation of the floating point operations is 41 times slower than the hardware. These results are also shown in the graph in Figure 3.

	Instr. Executed	*Instr. Executed (-O3)*
FP Hardware	12,024	4,015
FP SW (unpacked)	83,026	15,073
FP Software	336,254	165,010

Table 2: Floating point hardware performance versus floating point emulation performance.

Clearly even having five CPU cores operating in parallel is no match for a factor of 41 increase in performance for floating point hardware. These naive results are, however, somewhat unexpected. Clearly floating point arithmetic, particularly a multiply-accumulate operation, is not over 40 times more efficient than a software implementation. So investigation into the underlying implementation is in order.

Both implementations used the same exact source code shown in Figure 2, which was a simple loop repeated 1,000 times containing a multiply-accumulate operation. This is a simple benchmark, but one that contains a high ratio of floating point operations, in order to draw a fair comparison. Both were compiled with the Gnu gcc compiler version 4.2.2 using the standard libraries.

Upon closer investigation, it was revealed that the floating point emulation libraries perform a substantial amount of work keeping the data in a rigid floating point format. This format is the standard IEEE format with a single bit for the sign, eight bits for the exponent and the remaining 23 bits for the mantissa. A final extra leading mantissa bit is always assumed to be '1'.

These four fields are 'unpacked' before each floating point operation, giving the integer arithmetic units access to the individual fields for the calculations. In addition, after each operation is complete, the data is 'packed' back into the IEEE format. This pack / unpack combination is performed on each data item for each floating point calculation performed.

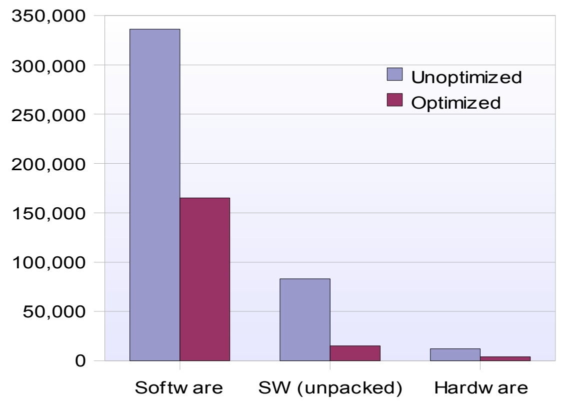

Figure 3: Floating point performance.

This packing and unpacking of data adds nothing to the computation and is used simply to keep data in a format favored by the floating point hardware, which is not in use. An alternative is to unpack the data once at the beginning of the calculation and pack it back again into the standard format when it is required by other hardware or routines such output.

The middle row in Table 2 gives the results for this 'unpacked' calculation. Unlike the naive implementation from the standard libraries, this version is approximately 7 times slower than the floating point hardware implementation in the unoptimized version, and 3.75 times slower than the floating point hardware implementation in the optimized version.

This result indicates that over 95% of the time in the emulation routines is spent packing and unpacking data.

Unfortunately, this type of code is very difficult for compilers to automatically optimize and currently has to be eliminated manually. However, the ability to achieve only a factor of 3.75 slowdown in a multicore environment that has five times the number of cores appears to support the idea that even for floating point hardware, more cores represent a higher performance implementation than special purpose hardware.

V. Conclusions

As more and more desktop, server and embedded processors move to multicore architectures, a more thorough study of multicore architectural issues should be undertaken. Most significant are the combined questions of how many cores should a design use and how large should those cores be.

The CPU cores used in today's designs appear to arise more from convenience than any other measurable factor. The most recent generation of cores is typically replicated in a device with little indication of other alternatives being explored. This is problematic because the microprocessor core of today has undergone a very long process of development involving many orders of magnitude increase in size and performance. The changes to these cores along the way had a single goal: to improve *serial performance* in a *single* core environment.

This focus on improving serial performance was understandable for single core devices. It preserved the programming model and kept legacy software operating on subsequent generations of devices. But once the break with uniprocessing has been made and the programming model broken, many of the architectural decisions made to create these cores come into question.

In this paper we explored some very early decisions in microprocessor history: the addition of a floating point unit. This occurred somewhere in the middle of this history of microprocessor development, approximately halfway between the original Intel 4004 and today's quad core devices.

Close examination indicates that even for simple code with very high proportions of floating point operations, the use of special purpose hardware for floating point operations may not be warranted. The ability to perform such operations in software using simpler integer operations appears to be more efficient in a multicore environment.

Of course, multicore architectures are new and there is much to be learned. In particular software issues are more important in multicore devices than in traditional single core architectures. The ability to parallelize software across a large number of cores will be the key to successfully exploiting the potential high performance of multicore devices. As work on tools, algorithms, and programming techniques progress, the parameters which define emerging multicore architectures will become more sharply defined.

But the situation is somewhat unusual. Multicore devices of many types are already in production, even without basic software support. A large swath of the semiconductor industry is now dependent on significant breakthroughs in application level software which already appear to be overdue.

REFERENCES

[1] “The Future of Microprocessors”, David Patterson, U. California at Berkeley, June 2001. http://www.cs.berkeley.edu/~pattrsn/talks/NAE.ppt

[2] Kunle Olukotoun and Lance Hammond, "The Future of Microprocessors", ACM Queue, Volume 3, Number 7, September 2005, pages 26-34.

[3] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, K. Scott Hemmert, "Architectural Modifications to Improve Floating-Point Unit Efficiency in FPGAs", International Conference on Field Programmable Logic and Applications (FPL) , 2006. Aug. 2006 Pages 1 - 6.

[4] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, K. Scott Hemmert, "Embedded floating-point units in FPGAs", Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA), Monterey, California, February 22-24, 2006, pages: 12 - 20.

[5] Krste Asanovic, Ras Bodik, Jim Demmel, John Kubiatowicz, Kurt Keutzer, Edward Lee, George Necula, Dave Patterson, Koushik Sen, John Shalf, John Wawrzynek, and Kathy Yelick, "The Landscape of Parallel Computing Research: The View from Berkeley 2.0", Manycore Computing Workshop, June 2007, http://science.officeisp.net/ManycoreComputingWorkshop07/Presentations/David%20Patterson.pdf

[6] Rey Archide, "The Microblaze v5.0 Soft-Processor Core: Optimized for Performance", Xilinx Embedded Magazine, November 2006, pages 18 - 21.

[7] "Intel Consumer Desktop PC Microprocessor History Timeline, http://www.intel.com/pressroom/kits/core2duo/pdf/microprocessor_timeline.pdf

[8] Cmpware, Inc. http://www.cmpware.com/, 2008.