DECODING RE-CONFIGURABLE PROCESSOR ARCHITECTURES

Rohit Kumar
Department of Computer Science
Government College, Raipur Rani, Panchkula
rohitbhullar@gmail.com

ABSTRACT

There have been significant advancements in the designs of processors and the tools used to produce and modify them. The issues related to using wall, dark silicon, and hardware reconfiguration along with code change have motivated progress in this area. These factors are crucial for microprocessor-based electronic projects and products that need significant, real-time, and mobile computational power in today's world. Efforts have also been made to lower the communication overhead in key areas of code through the use of specialized hardware and significantly decrease the fetch decode cycle. Recent studies have demonstrated that addressing both types of communication delays and addressing the issue of dark silicon can enhance power efficiency by 7 to 1000 times.

Key Words: Dark Silicon, utilization Wall

1. INTRODUCTION:

1.1 Concept of Dark Silicon

Dark silicon is the unused portion of a silicon chip due to certain limitations. Dark silicon may be present due to various causes such as improper chip programming or insufficient energy to activate transistors. This thesis focused on investigating and addressing the issue of dark silicon caused by power limitations. A specialized conservative approach has been developed to address the issue of 'Dark Silicon'.

1.2 Concept of Utilization Wall

The idea of utilization asserts that as each new process generation is introduced, the number of transistors that can operate at full frequency decreases exponentially because of power limitations. Because of this, a significant amount of chip area remains unused. A utilization wall is formed to divide the silicon chip into its used and unused parts. As per Moore's law, the quantity of transistors increases exponentially every year. This rule has held true for many decades, but due to power limitations, individuals cannot run the transistor count at maximum frequency, leading to underutilization. This issue of

not being fully utilized has been addressed in this paper.

In 1965 Gordon E. Moore introduced a principle stating that the number of transistors on integrated circuits will double roughly every two years throughout the history of computing hardware. However, with the breakdown of Dinnard theory [7], this principle no longer applies to the newest ultra scale integrated circuits. Dennard's scaling rules state that the relationship between voltage and current should be directly related to the size of a transistor, meaning that the power consumption, which is the result of multiplying voltage and current, will be directly related to the size of the transistor. This characteristic means that reduced MOSFETs, CMOS will use less power, and laid the foundation for Moore's Law. However, this scaling theory proved unsuccessful, resulting in the development of multi-core processor architectures. It occurred due to the power resources staying constant and being unable to switch the chip to full frequency. Despite the fact that the computing abilities of a fixed-size chip are rapidly growing at a rate of 2.8 times per process generation, this is due to enhancements in maximum transistor count (2 times) and improved transistor frequencies (1.4 times), the energy required is also increasing.

2. BEHAVIOR AND ADVANCED TOOLS

2.1 C-core Mimics Software Code

C-cores actually mimics the software code that is synthesized on it. During synthesis all of the instruction given in the program are hardwired in c-cores with all associated operands. E.g Consider the following for loop:

```
for ( i=0; i<=10; i++)
{ a=a+i;
}
```

Figure 1 presents an abstract hardware synthesis of the for loop. The variable (i) used in the loop has taken the shape of a fixed register of four bits, and will be initialized with 0, the comparison in the for loop has been replaced by a fixed logic comparator, addition had been implemented by the adder circuit, the parenthesis are implied by the arrow among the for loop components. One more register names (a) will be created to hold the value of variable (a). The number of bits taken by the (a) register will be determined by the initial value of (a) and plus execution of for loop.

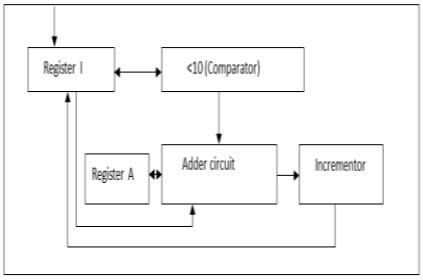

Figure 1. Synthesis of for loop in c-core.

2.2 Execution Model

Automated tools are used to profile work load. At design time, automated tool clusters c-cores on the basis of profiling of typical smartphone work-loads, examining both control flow and data movement between code regions. It places related c-cores on the same or nearby tiles, and in some cases, replicates them. At runtime, an application starts on one of the general purpose CPUs, and whenever the CPU enters a hot-code region, transfers execution to the appropriate c-core. Execution moves from tile to tile on the basis of the applications that are currently active and the c-cores they use. Coherent caches let data be automatically pulled to where it's needed, but data associated with a given c-core will generally stay in that c-core's L1 cache.

2.3 Synthesis of Android stack on a Core

Android is an open-source mobile software stack developed by Google that features a Linux kernel, a set of application libraries, and a virtual machine called Dalvik. User applications, such as those available in the application store, run on top of the Dalvik virtual machine.

It has been found that Android is well-suited for c-cores for several reasons. First, although many applications are available for download, Android has a core set of commonly used applications, such as a Web browser, an e-mail client, and media players. Typically, hot code is concentrated in the application libraries, the Dalvik virtual machine, and a few locations in the kernel. Because the hot code is well concentrated, targeting all these components with c-cores lets user attain high coverage over the source base and a significant impact on overall energy usage. Although c-cores support patching, which reduces the impact of post-silicon source base modification, users are also aided by smartphones' short replacement cycle (typically every 2 to 3 years), which lets smartphone chip designers deploy new c-cores to target new applications. The c-cores interface lets Android phone designers remove c-cores from their designs without impacting code compatibility.

2.4 Patching Support

To remain useful as new versions of the any platform emerge, the proposed architectures c-cores must adapt accordingly. To support this, c-cores include targeted re-configurability that lets them maintain perfect fidelity to source code, even as the source code changes. The adaptation mechanisms include redefining compile time constants in hardware and a

general exception mechanism that lets c-cores transfer control back and forth to the general-purpose core during any control flow transition. Adding this re-configurability increases the energy and area needs for c-cores, but significantly improves the span of years over which c-cores can provide energy savings.

The processor will contain many different c-cores, each targeting a different portion of the platform used. Designing each c-core by hand isn't tractable, especially because software release cycles can be short. Instead VHDL, Verilog tool-chains are used that converts arbitrary regions of code into c-core hardware.

The tool-chain first identifies the hot codes in the applications. The compiler then generates Verilog code for the control unit and data path that closely mimics those hot codes. The compiler also generates function stubs that applications can call in place of the original functions to invoke the hardware.

Finally, the compiler generates a description of the c-core that provides the basis for generating patches that will let the c-core run new versions of the same functions. The close mapping between the compiler's intermediate representation and the hardware is essential here: small, patchable changes in the source code correspond to small, patchable changes in the hardware.

Because c-cores focus on reducing energy and power consumption rather than exploiting high levels of parallelism, they can profitably target a much wider range of C constructs. Although conventional accelerators struggle to speed up applications with irregular control and limited memory parallelism, c-cores can significantly reduce the energy and power costs of such codes.

2.5 Implementing Android Graphic Library

Figure 2 shows a tile used to implement graphic function for Android based phone. The tile contains nine c-cores targeting important functions from the Android code base. Of these nine targeted functions, seven come from libskia, a 2D graphics library that provides compositing, rendering, and geometry calculations for most Android applications. The other two come from a JPEG decompression library and a fast Fourier transform (FFT).

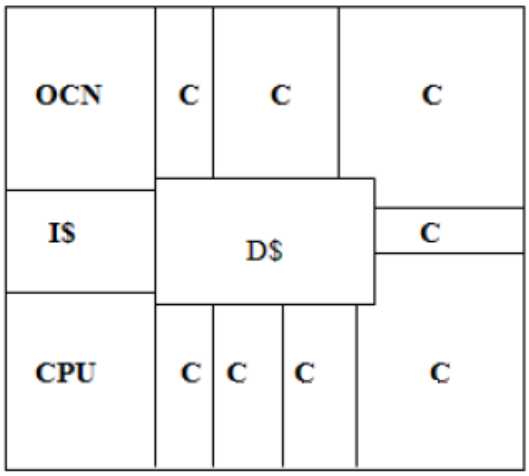

Figure 2: Single Tile targeting Android Graphic Library Functions.

Table 1 presents basic image processing functions for Android software stack. The tile in Figure 2 contains c-cores for these nine functions, many of which are from libskia, Android's 2D graphics library. The c-cores range from 0.024 mm^2 to 0.168 mm^2 and cover 10 to 90 percent of execution. Other tiles will have c-cores for other Android functions.

No.	Description	Android function	Dynamic execution coverage (%)	No. of static instructions	Size (mm^2)
1.	Dithering function	S32A_D565_Opaque_Dither	2.78	80	0.55
2.	Down sampling	S32_opaque_D32_filter_DXDY	2.20	86	0.070
3.	Bit-block image transfer subroutine	S32A_Opaque_BlitRow32	1.15	96	0.024
4.	Render with overlay	Sprite_D16_S4444_Opaque::blitRect	1.11	96	0.059
5.	Saturating down sampling	Clamp_S16_D16_filter_DX_shaderproc	0.80	97	0.063
6.	Fast Fourier transform (FFT)	fft_rx4_long	0.76	138	0.066
7.	Image conversion algorithm	ycc_rgba_8888_convert	0.61	92	0.046
8.	Lempel-Ziv decompression routine for GIF files	DGifDecompressLine	0.59	334	0.168
9.	Image format conversion for dithering	Sample_Index_D565_D	0.57	67	0.032

Table:1. size and dynamic execution coverage of c-cores for specific functions.

Most of the graphic library can be implemented with a single tile and is sufficient for graphic rendering. If the library contains many modules then tile with more number of cores must be used. Even if the graphic library is too large then more than one tile can also be used.

2.6 Energy Consumption Per Instruction

On average, each c-core occupies 0.064 mm^2 and runs at 1,568 MHz. Together, all the c-cores occupy 58 percent of the tile's area. The code they execute accounts for a total of 10.6 to 89.9 percent of execution for the benchmarks. Each of the core has 16 tiles and executes a full protocol stack with associated applications. Figure 3(b) shows the projected energy savings in processor and the origin of these savings. The savings as mention already come from two sources. First, c-cores don't require instruction fetch, instruction decode, a conventional register file, or any of the associated structures. Removing these reduces energy consumption by 56 percent. The second source of savings (35 percent of energy) comes from the specialization of the c-cores' data path.

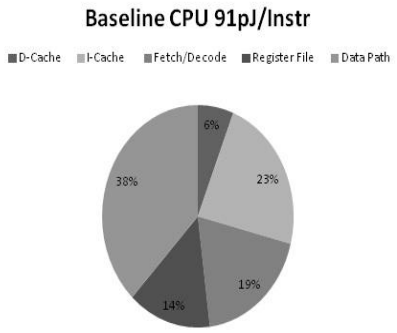

Figure 3(a).: Power consumption

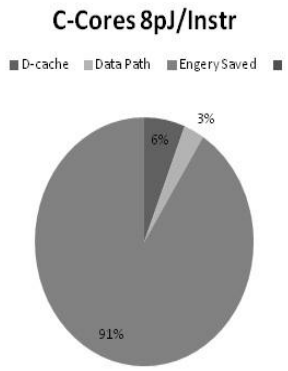

Figure 3(b).: Power Saving

The energy saved needs to be capitalized and conservatives cores effectively utilizes these energy savings. Due to reduction into these overheads average energy consumed per instruction drops exponentially. Maximum code base needs to be converted into c- cores to effectively utilize savings provided them. If less amount will be executed on c-cores then energy consumption will increase and is not desired. Figure 3(a) present's energy consumption in conventional baseline CPU, which consumes lots of energy as compared to conservative cores. Energy consumed per instruction will be very large due to various overheads in baseline processor.

3. OTHER CONSIDERATIONS AND IMPROVEMENTS

In recent time some processor has been developed which has removed the problem of dark silicon and utilization wall. In the proposed system conservative cores and accelerator has been used. Software code is

synthesized in these c-cores and accelerator and automated tools are required for synthesis. Such automatic tool-chain has been developed which automatically generates placed-and-routed c-core tiles, given the source code and information about execution properties. The cycle and energy accurate simulation tools have confirmed the energy savings provided by c-cores. Currently the work is being done on more detailed full system emulation to improve the workload modeling so that tools can finalize the selection of c-cores that will populate processors dark silicon. In parallel with this effort, work is being done physical design.

Identification of code for conservative core is very crucial and difficult task. Manual configuration of conservative cores is not possible so, automated tool have been devised to accomplish synthesis. Presently work is being done on development of optimized tools for synthesis. In past accelerator were used for improving efficiency. Both high level synthesis tool and accelerator has been discussed in next subsection.

3.1 Accelerators

Software code always executes at lower speed then the code executed by the hardware directly. Accelerators has been used in past to boost up performance of execution by converting some codes into hardware. Accelerators can be customized by the programmer for specific use. Here accelerators are actually hardware that offers very fast computation. Parts of the algorithm that can be parallelizable are usually converted in to accelerative hardware. In addition, the codes which are large and are used frequently are converted into accelerators to gain performance efficiency.

One extreme example of such an accelerator is Anton processor, which attains 100 times or more improvement in molecular-dynamics simulation versus general-purpose supercomputer machines. Many functions like baseband processing, 3D graphics, or video decoding or encoding are usually converted in to accelerators. The challenge in creating accelerators is in reorganizing the algorithm to achieve parallel execution. Being able to do this effectively depends on the availability of exploitable parallelism in the algorithm and the ability to expose this parallelism in the form of an accelerator circuit without errors or excessive effort, complexity, or cost.

In particular, creating accelerators for irregular code that’s difficult to analyze or lacks parallelism is very challenging. The conservation cores that comprise the proposed system have different, but related,

underlying goals compared to accelerators. Fundamentally, conservation cores (c-cores) focus on reducing the energy of executing code, even if they only modestly improve the resulting execution time. As a result, c-cores don't rely on parallelization technology for a successful outcome, they can target any code in which sufficient execution time is spent. Because of this, a far greater percentage of code can be turned into c-cores than can be transformed into accelerators.

In the commercialized version of the proposed architecture accelerator can be used for performing some fixed tasks. The code that can be parallelized is also a good candidate for conversion into accelerator. The remaining un-accelerable, irregular needs to be tracked and can be synthesized into conservation cores. This distinction forms some of the key differences between c-cores generation and conventional high-level synthesis technology, which focuses on generating accelerators from parallelizable code.

3.2 High-level Synthesis Tools

High level synthesis (HLS) tools are used to create accelerators that improve the execution performance of critical algorithms by implementing these algorithms in integrated circuits. HLS research has a long and rich history, which has resulted in the availability of several commercial tools, including AutoESL'sAutoPilot, Cadence's C-to-Silicon Compiler.

Because these tools seek to infer parallel execution from serial code, they have many of the same limitations that parallelizing compilers suffer from—namely, the difficulties of analyzing pointers in free form code, extracting memory parallelism, and extracting and formulating efficient parallel schedules for the operations in critical loops. These are difficult tasks that are generally NP-complete or worse in complexity and such difficult code sections are addressed in special way by the automated tools used for synthesis.

To address the parallelization challenges, HLS tools have removed some of features (for example, no pointers, no dynamic memory allocation, or no goto's) or relying on user-transformed code or programs for guiding the tool in generating output with good results. Typically, operating-system code and I/O code are considered synthesizable with lots of difficulty and either the HLS tool ignores this code or the user must comment it out or put it to a different part of the system.

Because the proposed system targets a system with millions of lines of difficult to parallelize code, including the various

kernels of different platforms, so the tool-chain must automatically and successfully reduce the energy of large bodies of non-parallelizable code without user intervention. The code base is too large to afford manual intervention, and is constantly evolving.

The tool-chain doesn't need user programs for effective transformation or require any source code modification to remove unsupported constructs. Additionally, the patching mechanism lets the developer support changes in the underlying source base, which lets the c-cores evolve as the targeted software changes.

To convert arbitrary region of program code good synthesis tool must be used. The tools used must be able to identify hot code in large portion of code base. Additionally converting code that doesn't execute frequently into c-cores will have sever effect on performance of the system.

4. CONCLUSION

The proposed prototype has been designed to obtain lots of energy savings over conventional processor design. The proposed prototype efficiently reduces the impact of dark silicon and utilization wall. Conservative cores have been introduced which are actually responsible for removing the problem of utilization wall and dark silicon.

Each processor tile contains variant number of conservative cores and has all other circuitry like general CPU, instruction cache and data cache etc. for executing any kind of software code. Identification of hot code is must for achieving good efficiency and power savings. Automated tools must have to be used as manual synthesis is not possible for large code bases. Work is being done on optimization of automated tools to gain maximums efficiency.

Actual synthesis is also very difficult as the prototype needs very large area. The tools like verilog must be programmed carefully and must be thoroughly tested for good synthesis.

5. REFERENCES:

[1] Mombert, G., http://www.digitaltrends .com/mobile/what-is-asmartphone/ [online], last seen dec, 2010.

[2] Johnny John and Chris Riddle, "Smartphone Power", proceedings of DAC, Anaheim, California, USA, pp. 935-936, 2010.

[3] Vinay Mehta, http://berylsystems.com /smartphone.pdf [online], seen oct, 2010.

[4] Steven Cavanagh and Yingxu Wang, "Design of a Real-Time Virtual Machine

(RTVM)", proceedings of Electrical and Computer Engineering 2005 Canadian Conference, pp. 2021-2024, May, 2005.

[5] Omar A. Fres and Ignacio G. Alonso, "Rovim: A Generic and Extensible Virtual Machine for Mobile Robots", Fifth International Conference on Systems held in USA, pp. 37-40, 2010.

[6] Michael Bedford Taylor, "The Raw Microprocessor: A Computational Fabric For Software Circuits and General-Purpose Programs", IEEE proceedings, pp. 24-35, 2002.

[7] Robert H. Dennard, "Design of lon-Implanted MOSFET's with Very Small Physical Dimensions", IEEE Journal of Solid-State Circuits, vol. SC-9, pp. 256-268, October 1974.

[8]. AsafAshkenazia, DimitryAkselrodb and Yossi Amona, "Platform Independent Overall Security Architecture in Multi-Processor System-on-Chip ICs for Use in Mobile Phones and Handheld Devices", proceedings of World Automation Congress (WAC), vol.33, issue no.5-6, pp. 407-424, 2007.

[9]. Meira Levy, PeretzShoval, BrachaShapira, Aviram Dayan and MeytalTubi, "Task Modeling Infrastructure for Analyzing Smartphone Usage" Ninth International Conference on Mobile Business / 2010 Ninth Global Mobility Roundtable, pp.264-271, 2010.

[10]. Bhullar, Rohit K., et al. "Intelligent stress calculation and scheduling in segmented processor systems using buddy approach." Journal of Intelligent & Fuzzy Systems 32.4 (2017): 3129-3142.

[11]. Taneja, Kavita, HarmunishTaneja, and Rohit K. Bhullar. "Cross-platform application development for smartphones: Approaches and implications." 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom).IEEE, 2016.

[12]. Arora, Swinky, et al. "Novel stress calculation in parallel processor systems using buddy approach with enhanced short term CPU scheduling." Proceedings of the International Conference on Communication and Computing Systems (ICCCS 2016). 2016.