# Compilation Accelerator on Silicon

Venkateswaran Nagarajan*, Vinesh Srinivasan‡, Ramsrivatsa Kannan‡, Prashanth Thinakaran‡, Rajagopal Hariharan‡, Bharanidharan Vasudevan‡, Nachiappan Chidambaram Nachiappan†, Karthikeyan Palavedu Saravanan†, Aswin Sridharan†, Vigneshwaran Sankaran†, Vignesh Adhinarayanan†, V.S.Vignesh†and Ravindhiran Mukundrajan†
*Director, ‡WARFT Research Trainee, †Previously affiliated with WARFT
Waran Research Foundation [WARFT], India
Email: waran@warftindia.org

***Abstract*—Current day processors utilize a complex and finely tuned system software to map applications across their cores and extract optimal performance. However with increasing core counts and the rise of heterogeneity among cores, tremendous stress will be exerted on the software stack leading to bottlenecks and underutilization of resources. We propose an architecture for a Compilation Accelerator on Silicon (CAS) coupled with a hardware instruction scheduler to tackle the complexity involved in analyzing dependencies among instructions dynamically, accelerate machine code generation and obtain optimum resource utilization across the cores by effective and efficient scheduling. The CAS is realized as a two-level hierarchical subsystem employing the Primary Compiler on Silicon (PCOS) and Secondary Compiler on Silicon (SCOS) with the hardware instruction scheduler as an integral part of it. A comparative analysis with the conventional GCC compiler is presented for a real world brain modeling application and higher instruction generation rates along with improved scheduling efficiency is observed resulting in a corresponding increase in resource utilization.**

***Keywords*-Heterogeneous Multi-Cores; Hardware Compiler; Hardware Scheduler;**

## I. INTRODUCTION

The need for micro-architectural energy efficiency along with increased performance has spurred the growth of heterogeneous multi-core architectures [1] [2]. Parallelism in all forms have been embraced in an attempt to increase the throughput of the system. While plenty of efforts are being dedicated to accelerate the execution of various applications, an equivalent effort to speed up code generation and scheduling has not been observed so far. Currently, a complex and finely tuned system software is used to map applications across the different cores based on their computational requirements [2] and thus, effectively exploit the potential of the underlying silicon. However, due to deployment of hundreds of specialized functional units across many cores, there is a need for quicker dependency resolution and faster instruction issue. This will place tremendous stress on the system software encompassing the likes of compiler and instruction scheduler. To address this, the idea of a single instruction set, and thereby an opportunity to provide a scalable compiler across cores has previously been proposed in [3]. However, with large number of cores and disparate functional units inside each core, the instruction issue rate of a software compiler will not suffice.

In an attempt to address this bottleneck, an architecture for a Compilation Accelerator on Silicon (CAS) is proposed. The CAS is a hardware accelerator that will co-exist with the existing software ecosystem and will assist the system software in:

- tackling the complexity of issuing instructions in parallel
- increasing the instruction issue rate of dependent as well as independent instructions
- performing parallel mapping of multiple instructions belonging to different applications within and across cores.

The CAS performs the arduous task of analyzing the dependencies among assembly level instructions and generating machine level instructions. It can be realized as a two-level hierarchical architecture, as shown in Figure 1.

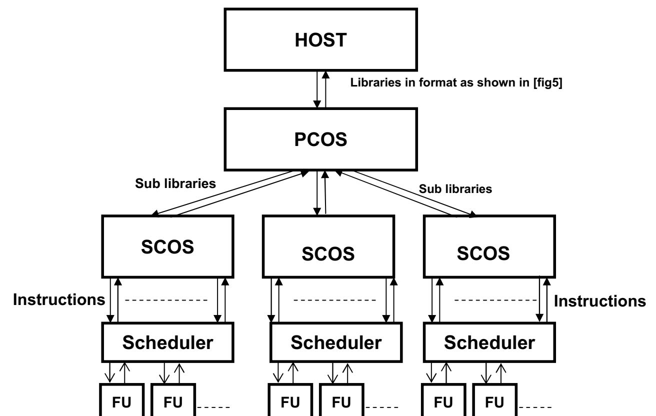


Fig. 1. Hierarchical representation of CAS

To further complement the CAS, we also propose the design of a hardware scheduler that will help improve resource utilization. The scheduling of instructions also plays a pivotal role in keeping the vast underlying resources busy and thereby increase throughput. The hardware scheduler is closely integrated with the CAS. The instructions are scheduled based on heuristics that will improve resource utilization and thus effectively exploit the potential of the architecture. The hardware scheduler is envisioned as a micro-programmed engine, which provides the user with the opportunity to program heuristics suitable to his application.

This paper focuses on the functional architecture of the code generator and scheduler. The detailed RTL level description of each functional block utilized in the architecture has been discussed separately in [4]. The CAS-hardware scheduler

978-0-7695-4767-1/12 $26.00 © 2012 IEEE
DOI 10.1109/ISVLSI.2012.76



IEEE computer society

subsystem has been verified and validated by means of Verilog simulations. A detailed comparison of the time taken to compile a real-world brain modeling application by a conventional GCC compiler and Compilation Accelerator on Silicon (CAS) is presented in this paper. Section II discusses the architecture of the Dependency Analyzer. The hardware scheduler and its heuristics are elaborated in Section III. The machine code generator is presented in Section IV. We discuss the obtained results in Section V and conclude in Section VI.

## II. Dependency Analyzer

Current day many core processors are designed to support out-of-order execution [5] to make the underlying hardware utilize the stall time for compute instructions. For such a scenario, ideally there should be an efficient technique for ordering instructions in the instruction queue, as well as, to differentiate among those instructions whose dependencies were resolved and those whose dependencies were not. This analysis of dependency among instructions and its associated data can be clearly understood by the following example. Consider an example with four different instructions: I1, I2, I3, I4. Let I3 depend on I1 and I2, while I4 depends on I2 only. Assume the current timestamp to be 126. Let us assume that I1 and I2 have been scheduled to its respective functional units execution at $121^{st}$ and $123^{rd}$ timestamps respectively. Calculating the time at which I3 can be scheduled to its respective functional unit:

*Time to execute I1 = Delay of multiplier unit = x*
*Time to execute I2 = Delay of comparator unit = y*
*Time at which I3 can be scheduled to its functional unit*
*= max(126, 121 + x, 123 + y)*

Keeping these functionalities in mind, we evolve the architecture of dependency analyzer which is depicted in Figure 2. The dependencies across instructions is stored in the instruction detail table along with instruction IDs marked as I1,I2, I3 and their sources are searched (Search unit 1.1 and Search Unit 1.2). Instruction detail table is discussed further in Section 4. The search unit employs graph traversal technique and their cell based pipelined architecture is given [6]. Apart from that, to set timestamp of the child instruction, latency of the underlying functional units which executes the parent instructions is needed and is stored in a separate table (not shown in Figure 2). For this purpose, two search units (Search unit 2.1 and Search Unit 2.2) are employed. More details on the tables are discussed in Section 4.

At the end of this operation, the output obtained will be the time stamp at which the parent instructions have been scheduled to their underlying functional units. This obtained timestamp has to be added to the latency of the respective functional units so as to obtain the final timestamp by which the instructions would have completed its execution. Suitable adder units [6] are provided for this operation. At the end of the execution, the final timestamps obtained from Search unit 1.1, Search unit 1.2, and the current timestamp are compared and the maximum among the three values (max finder Unit)

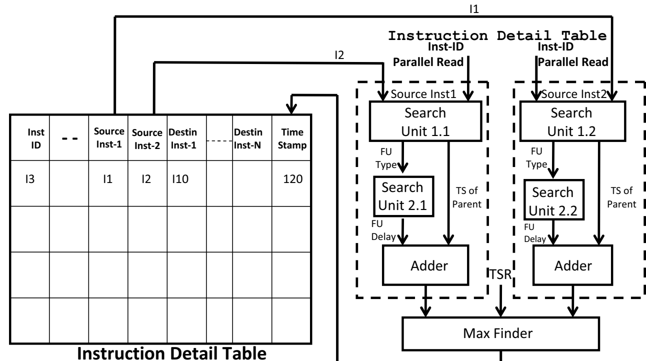


Fig. 2. Functional Architecture of the Dependency analyzer

[6] should be the timestamp at which the current instruction should be scheduled.

Similarly, the dependencies are analyzed for each instruction. The architecture of the dependency analyzer is designed to give priority to independent instructions and thereby increase resource utilization. The dependency analyzer analyzes the entire set of instructions and resolves the dependencies across them. The scheduler whose architecture is about to be seen in the next section, schedules instructions which have either no dependencies or those instructions whose dependencies are already resolved. Hence, the information provided by the dependency analyzer is vital for the scheduler for achieving better performance and thereby increased resource utilization. Dependency analyzer proposed includes exclusive hardware graph traversal units for analyzing dependencies across instructions unlike in any previous works [7].

## III. Hardware Scheduler

Conventional software scheduler uses scheduling algorithms such as round robin or pre-emptive scheduling. But, in case of a heterogeneous multicore processing environment, the scheduler should govern instructions of multiple applications that have to be scheduled to multiple cores effectively. In such a case, the software scheduler becomes an overhead in scheduling the large number of instructions at a rate catering to the needs of such instruction hungry processors. Hence, for high performance heterogeneous multicore processors, a hardware based scheduler becomes inevitable.

The proposed hardware scheduler, shown in Figure 3, is tightly coupled with both the levels of hierarchical CAS. It interacts with the CAS and utilizes its tables to effectively map the generated instructions. The hardware scheduler is also developed using higher level functional units called as Algorithm Level Functional Units (ALFUs) which results in its improved performance. The Algorithm Level Functional Unit is a superset of scalar units and higher level functional units [8].

The instructions generated by the code generator are scheduled to the underlying FUs by the hardware scheduler. The

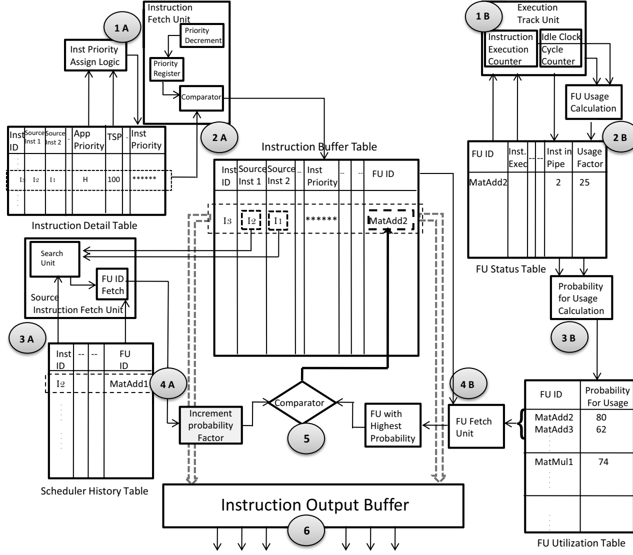


Fig. 3. Architecture of Hardware Scheduler

hardware scheduler takes into account the numerous parameters that are critical for the performance of a processor. For example, the hardware scheduler not only considers an application's priority for scheduling instructions, but also considers the optimal resource utilization of the functional units and load imbalance. The hardware scheduler is undoubtedly the nucleus of the code generator concerned with the overall performance of the processor. Some of the parameters taken into consideration by the hardware scheduler are functional unit usage, number of output buffer stages filled, number of instructions pipelined in functional unit, source instruction dependencies and application priority. The various tables its sizes and the devices utilized in hardware scheduler are shown in Table 1. The scheduler architecture is developed using simple heuristics that impacts scheduling of instructions under various circumstances as listed below:

### *A. Resource Utilization based Scheduling*

Uneven distribution of load across Functional Units (FU) causes over utilization of some FUs. This creates thermal hot spots which could lead to breakdown of processors. Therefore, load balancing across FUs is necessary. To achieve this, we monitor the execution status of each FU and compute its usage as follows:

$$\text{FU Usage} = \text{N} - \frac{\text{C}_{idle} \times \text{Latency}}{100} \tag{1}$$

where N denotes number of instructions executed, Number of clock cycles FU remain idle denoted by $\text{C}_{idle}$, Latency denotes latency of particular FU

The components in the architecture responsible for this computation are: (i) execution track unit [1B][Figure 3], which uses a set of HW counters and (ii) FU usage calculator unit [2B][Figure 3], composed of subtractor and divider ALFUs.

TABLE I
TABLE SIZES AND DEVICE COUNT OF THE SCHEDULER

| Table Name | Sizes | ALFU | Device Count |
|---|---|---|---|
| Library Address Table | 4 KB | CLA | 446 |
| Library Detail Table | 4 KB | Multiplier | 2000 |
| Library Status Table | 4 KB | NRD | 1792 |
| Sub-Library Address Table | 100 Bytes | Comparator | 688 |
| Instruction Status Table | 16 KB | MatMul | 19790 |
| Instruction Detail Table | 142 KB | MOA | 4154 |
| Instruction Buffer Table | 16 KB | Inner Product | 30934 |
| FU Status Table | 0.5 KB | Graph Traversal | 6672 |
| Scheduler History Table | 166 KB | Sorter | 11008 |
| FU Utilization Table | 192 Bytes | Max/Min Finder | 3388 |

Based on FU usage, probability factor for each FU is calculated as shown below. Here, probability factor is a term that indicates the probability that a given FU is chosen for scheduling.

$$P_F = \frac{100 - \text{FU Usage}}{50} + \frac{N_{max}\text{-}N_{current}}{25} - \frac{T_B - T_F}{25} \tag{2}$$

where $P_F$ denotes Probablity factor, $N_{max}$ is maximum number of instructions in FU pipe, $N_{current}$ denotes number of current instruction in pipe, $T_B$ and $T_F$ represents total and filled output buffer stages respectively.

The probability factor also takes into consideration the number of instructions currently in the FU pipe and number of o/p buffer stages filled. In the architecture, 'probability for usage calculator' unit [3 B][Figure 3], composed of multiplier and multiple operand adder ALFU, computes this term. This unit gets its input from the FU status table. Also, the scheduler maps a child instruction to same FU as the parent instruction to reduce communication overhead. For this purpose, Scheduler history table stores the history of each instruction. An instruction fetch unit, which uses an array of comparators, fetches information of parent instructions from this table.

### *B. Pipeline Stall based Scheduling*

Whenever pipeline stall occurs in a Functional Unit, the execution track unit [1B][Figure 3] keeps track of that Functional Unit's idle clock cycles. Depending on the number of instructions executed and idle clocks, usage of a particular FU is calculated. The FU which was idle for more clock cycles due to pipeline stalls is given a higher priority for execution of subsequent instructions.

### *C. Application Priority based Scheduling*

When multiple applications gets executed in the same core simultaneously [9], then the instructions of a higher priority application needs to be executed first. However, in this case, it is also important to ensure that instructions of a low priority application, which may be waiting in the schedule queue for a comparatively longer time, are eventually scheduled. A logic, that considers both application priority and assigned time stamp of instruction, has been developed as shown below:

$$\text{Inst}_{priority} = \text{App}_{priority} + (\text{TS}_{current} - \text{TS}_{assigned}) \quad (3)$$

where $\text{Inst}_{priority}$ and $\text{App}_{priority}$denotes Instruction and Application priority.$\text{TS}_{current}$ and $\text{TS}_{assigned}$ are current and assigned time stamps respectively.

Thus, the instruction having an older time stamp and a higher instruction priority increases its chances of getting scheduled. The priority of an instruction's application and its assigned time stamp are fetched from instruction detail table. The instruction priority assign logic unit [1A, 2A] [Figure 3], made of adder and subtractor units [8], takes this input and assigns the priority for each instruction.

### D. FU assigning Logic

Finally the probability factor computed based on parent instruction's history [3A,4A] [Figure 3] and FU usage[4B] [Figure 3] is compared and the FU with the higher probability factor is selected using a comparator ALFU. The instruction is finally mapped onto the selected FU. Existing hardware schedulers [10][11] lacks heuristics for optimal resource utilization of functional units. Though [11] does load balancing, it is achieved by naive process such as task stealing and has no specific units to track the instruction execution unlike our hardware scheduler.

## IV. Machine Code Generator

To tackle this increasing complexity of future generation processing cores, the CAS is designed so as to generate and issue instructions by which the architectural potential of the underlying units can be exploited efficiently. The compiling framework that has been proposed in this paper is hierarchical in order to distribute the complexity among the levels. Thus, a two level hierarchical architecture has been framed. They are Primary Compiler-on-Silicon PCOS and Secondary compiler-on-silicon SCOS.

The hierarchical CAS is also capable of performing dynamic compilation, run time dependency analysis, code scheduling and optimization decisions using the architecture of dependency analyzer shown in the previous section because static compilation suffers when it comes to non-stationary applications and runtime decision making systems.

The applications after going through the naive and initial stages of compilation like lexical analysis, syntax analysis and semantics at the host level, enter as inputs in the form of libraries to the first level of hierarchy- CAS. The libraries which are received as packets of instructions are broken down into sub-libraries at the PCOS and are scheduled to the second stage of hierarchy- SCOS. A PCOS may schedule and be in-charge of a number of SCOS(s). Out of the various SCOS present within the compiler, each may be employed to one or more cores within the architecture. SCOS assembles the sub-libraries, analysis the dependencies and issues ISA control words to trigger many Functional Units (FUs) in parallel. Since the ISA formation and scheduling rate depends on the

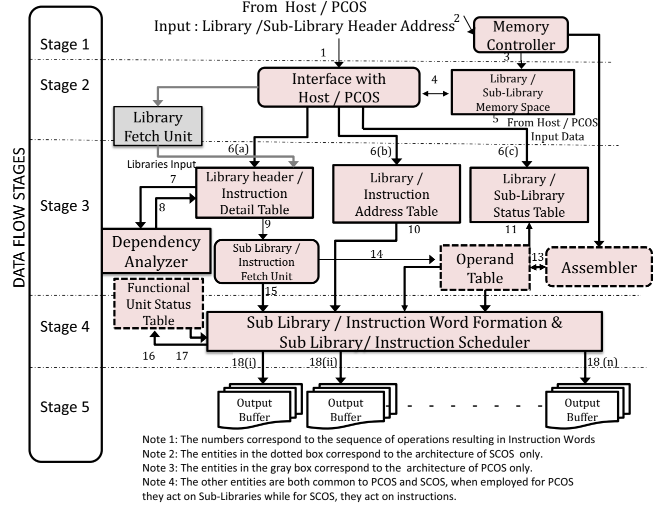


Fig. 4. Generic Architecture for PCOS and SCOS

hardware scheduler, it accounts directly to the resource utilization of the architecture and thereby performance acquired. All tables utilized by the CAS and its respective table sizes are shown in Figure 4 and Table 1 respectively.

### A. Primary Compiler-on-Silicon PCOS

Owing to the similarities in the architecture of PCOS and SCOS Figure 4 shows the generic architecture of both levels of PCOS. Each unit of the CAS architecture is responsible for a specific set of operations which is aided by the numerous tables CAS maintains. The Library Header input, sent from the host is received at the Library Header Buffer. The data from LIB Header is then used to fill the various tables in CAS. All available data are filled in the tables employed. These tables play an important role in maintaining the execution status and generating Sub-LIB words. LIB Address Table gets the address of the libraries from the SRAM Controller.

When a LIB is scheduled to CAS, its corresponding instruction and data packets are stored at the core's local SRAM banks. The addresses at which these are stored is intimated by the local SRAM Controller to the LIB Address Table. Similarly, Sub-LIB dependency in the instruction packet is also read and sent by the local SRAM Controller to LIB Header Detail Table for dependency analysis in the later stages.

Once the tables are filled, Dependency Analyzer of PCOS reads the LIB Header Detail Table to resolve the dependencies across the Sub LIBs present in a LIB and assigns timestamp. This timestamp value is maintained in a Time Stamp Register (TSR) present within Dependency Analyzer. This timestamp assigned, marks the scheduling and execution sequence of the Sub-LIBs. After timestamps are assigned, Sub-LIB Selection Logic selects an appropriate Sub-LIB based on assigned timestamp value from the LIB Header Detail Table and sends it to the scheduler.

**PCOS Tables**

Library Address Table

| Library Id | Sub-Library Id | Sub Library Instruction Address | Sub Library Data Address |
|---|---|---|---|

Library Detail Table

| Library Id | Sub-Library Id | No. Of Dependencies | Dep. Sub-Library ID | Computational Requirement | Priority | Time Stamp |
|---|---|---|---|---|---|---|

Library Status Table

| Library Id | Total No. Of Sub-Libraries | No. Of Sub-Libraries Executed | Sub-Library ID | Execution Status of Sub-Library | Execution Status of Library | Time Stamp |
|---|---|---|---|---|---|---|

**SCOS Tables**

Sub - Library Address Table

| Sub – Library Id | PCOS assigned timestamp | Sub Library Instruction Address | Sub Library Data Address |
|---|---|---|---|

Functional Unit Utilization Table

| FU Id | Probability Factor |
|---|---|

Instruction Detail Table

| Library Id | Instruction Id | Type of Instruction | Sou-rce 1 | Sou-rce 2 | Destin-ation 1 | Destin-ation 2 | Application Priority | Assigned Time Stamp | Instruction Priority |
|---|---|---|---|---|---|---|---|---|---|

Scheduler History Table and Instruction Status Table

| Library Id | Instruction Id | Type of Instruction | Sou-rce 1 | Sou-rce 2 | Destin-ation 1 | Destin-ation 2 | Instruction Priority | Execution Status | No. Of Inst present | No. Of Inst executed | Assigned FU ID |
|---|---|---|---|---|---|---|---|---|---|---|---|

Instruction Buffer Table

| Instruction Id | Type of Instruction | Sou-rce 1 | Sou-rce 2 | Destin-ation 1 | Destin-ation 2 | Application Priority | Assigned Time Stamp | Instruction Priority | FU ID |
|---|---|---|---|---|---|---|---|---|---|

Functional Unit Status Table

| FU Id | Status | Clock Assigned | Pipelined Delay | No of Inst Executed | No of Inst in Pipe | Pred Status | Usage Factor |
|---|---|---|---|---|---|---|---|

Fig. 5. Fields in PCOS and SCOS tables

| Core ID | Instruction ID | Source Inst 1 | Source Inst 2 | Destination 1 | Destination 2 | App Priority | Inst Priority | Assigned Time stamp | ALFU ID |
|---|---|---|---|---|---|---|---|---|---|

Fig. 6. Format of the machine level code word generated by the code generator. The word length of each field varies depending on the sizes of the tables in the CAS

Scheduler within PCOS is responsible for scheduling the Sub-LIB Header formats to the PCOSs. It is the heart of PCOS as it is responsible for the effecting task level parallelism by allocating the Sub LIBs to the multiple SCOSs under it. Scheduler uses the Sub-LIB status table to check if a Sub LIB has already been scheduled for execution. Except the status field, all other details are filled by the data available from LIB Header Buffer. The status of executing Sub-LIBs is communicated to it by the SCOS which execute those Sub-LIBs. Scheduling is based on the heuristics implemented as a part of the system libraries. When it has been decided as to which and where the Sub-LIB has to be scheduled, the Sub-LIB Header Format is assembled using LIB Address Table and sent to their respective SCOS Input Buffers.

### *B. Secondary Compiler on Silicon*

The Secondary Compiler-On-Silicon (SCOS) is mesh connected and operates in a parallel fashion to extract parallelism within the architecture. Thus, various SCOS functioning independently enable concurrent generation of instructions and their scheduling. While the PCOS maps application libraries to different SCOS, the SCOS executes the various general purpose instructions (within the sub-libraries) independent of each other by resolving their dependencies. The mesh topology of SCOS facilitates sharing the status of the different SCOSs.

The architecture of SCOS is very similar in its functional working to the PCOS, however with the exception that the SCOS works on instructions rather than Sub-Libraries. A simple addition in the architecture of SCOS compared to that

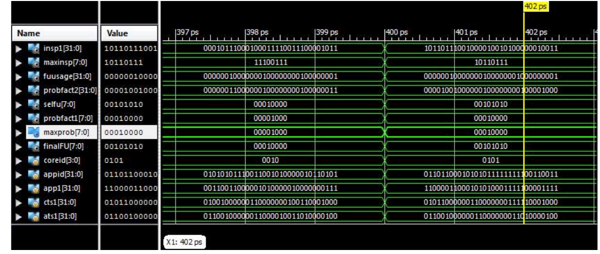


Fig. 7. Verilog Simulation of Hardware Scheduler

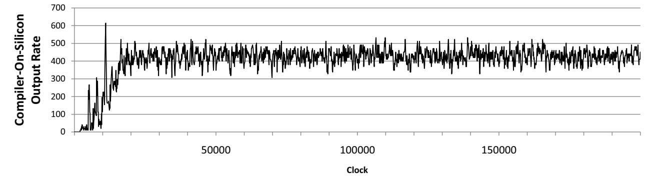


Fig. 8. CAS output rate for SPEC benchmark ASTAR

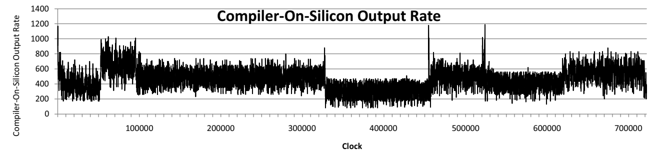


Fig. 9. CAS output rate for SPEC benchmark GCC

of PCOS is the inclusion of functional unit status table which helps the SCOS scheduler to distribute load evenly and to check if a particular functional unit is available for scheduling. Adding to that, there is an assembler which assembles the assembly level instructions (read form the Instruction Packets) and Operand Table with the address of operands (from the data packet address location). A table that is solely dedicated for this purpose is called instruction buffer table. Instructions to be scheduled at a particular time stamp are grouped along with their data addresses obtained from assembler to form an Instruction Word Figure 6, which is then sent to the cores.

## V. Results and Ananysis

This section explains the experimental setup and simulation framework of the proposed CAS. The CAS simulator [12] is a clock-accurate simulator used to implement, verify and validate the working of the proposed concept. The CAS is implemented as sub-simulator into the CUBEMACH simulator [12] which simulates heterogeneous functional units. The CUBEMACH simulator is set up with initial functional unit parameters consisting of various Algorithm Level Functional Units [8] (MatMul, MatAdd etc) and Scalars (Add, Sub, Mul, Div). This information is used to setup tables (such as Functional Unit Status Table) within CAS. Algorithms similar to ones in SPEC Benchmarks were used to test and verify the working of the CAS. We write micro-kernels that we believe are close representatives of the actual SPEC benchmarks and use the same names as the SPEC benchmarks for convenience.

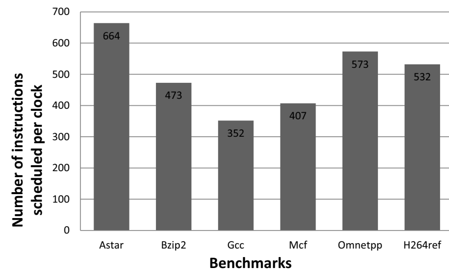


Fig. 10. Number of Instructions scheduled per clock for different Equivalent SPEC benchmarks

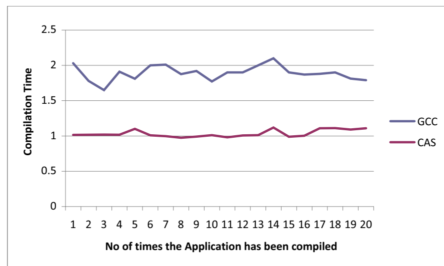


Fig. 11. Comparison between the time taken compile a multi million neuronal application in a conventional compiler and Compilation Accelerator on Silicon.

In Figure 8 and Figure 9, benchmarks are run sequentially in the simulator and corresponding scheduler output rate is recorded. The zoomed-in portion of the diagram shows the output rate for the benchmark GCC (Figure 9) which is a high instruction dependent application, whereas ASTAR (Figure 8) has large amounts of independent instructions, resulting in a constant output rate. The graph shows the efficiency of the compiler-scheduler in mimicking the characteristics of the algorithms and their dependencies.

Figure 10 shows the bar chart for the average number of instructions scheduled per clock corresponding to the type of benchmark application executed. As seen in the figure, for SPEC benchmarks like ASTAR, the average number of instructions scheduled is maximum compared to other benchmark sets. This is due the fact that ASTAR has large number of independent instructions to be scheduled every clock cycle. Graph based benchmark set of Bzip2 have low independent operations. Thus, interdependency across instructions leave CAS to schedule only limited number of instructions to functional units. Specific applications like GCC, MCF which are used to form high data dependent applications, have lesser instructions scheduled than any benchmark executed.

Figure 11 depicts the quantitative comparison between CAS and GCC with respect to compilation time of a multi million neuronal inter-connectivity prediction based application [13] [14]. The equivalent Benchmark Simulator [12] (a synthatic workload) of the application is run separately in a GCC compiler and CAS. The time taken for syntax and semantic analysis of the GCC compiler is identified and is eliminated from the total compilation time such that only the time taken for code generation is considered for comparison. It is evident from the graph that there is a marginal difference in compilation time for the same application compiled with two different compilers. This marginal difference plays a major role when compared in terms of number of clock cycles. Hence, for CAS the number of clock cycles taken to execute a particular application would be significantly lower when compared to GCC.

## VI. Conclusion

This paper proposes CAS, Compilation Accelerator on Silicon, a novel approach towards the design of a hardware compilation accelerator with an integrated scheduler. The CAS's hierarchical structure enables easy customization, and provides a dynamic and powerful architecture aware scheduling heuristics. Our evaluations show the potential benefits that can be obtained through the use of the accelerator. We believe our approach in using hardware accelerators for compilation and scheduling is essential for mitigating the software bottleneck involved in scheduling for complex heterogeneous architectures.

## References

[1] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *HPCA*, 2011, pp. 503–514.
[2] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *MICRO*, 2003, pp. 81–92.
[3] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *ISCA*, 2004, pp. 64–75.
[4] WARFT, "Rtl details of compilation accelerator on silicon," http://www.warftindia.org/cas/cas.pdf, Jun. 2012.
[5] L. Villa, R. Espasa, and M. Valero, "A performance study of out-of-order vector architectures and short registers," in *ICS*, 1998, pp. 37–44.
[6] N. Venkateswaran *et al.*, "Scoc ip cores for custom built supercomputing nodes," in *ISVLSI*, 2012.
[7] L. J. Carter, "Compiler and hardware predicated dependency analysis and scheduling," Ph.D. dissertation, University of California, San Diego, 2002.
[8] WARFT, "Alfu architectures and verilog simulations," http://www.warftindia.org/cas/alfu.pdf, Apr. 2012.
[9] N. Venkateswaran *et al.*, "On the concept of simultaneous execution of multiple applications on hierarchically based cluster and the silicon operating system," in *IPDPS*, 2008, pp. 1–8.
[10] P. Kuacharoen, M. Shalan, and V. J. Mooney, "A configurable hardware scheduler for real-time systems," in *ERSA*, 2003, pp. 95–101.
[11] S. Kumar, C. J. Hughes, and A. D. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA*, 2007, pp. 162–173.
[12] WARFT, "Cubemach design paradigm simulator," http://www.warftindia.org/CUBEMACH/CUBEMACH-2.2.tar.gz, Jun. 2012.
[13] N. Venkateswaran *et al.*, "Energetics based spike generation of a single neuron: simulation results and analysis," *Frontiers in Neuroenergetics*, vol. 4, no. 00002, 2012.
[14] A. Mohan, "The mmini-dass simulator and its application to visual pathway connectivity prediction," Thesis submitted to Waran Research Foundation, India, 2008.