Chen P, Zhang L, Han YH *et al.* A general-purpose many-accelerator architecture based on dataflow graph clustering of applications. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(2): 239–246 Mar. 2014. DOI 10.1007/s11390-014-1426-9

A General-Purpose Many-Accelerator Architecture Based on Dataflow Graph Clustering of Applications

Peng Chen[1,2] (陈 鹏), Lei Zhang[1] (张 磊), *Member*, *CCF*, *ACM*, *IEEE*
Yin-He Han[1] (韩银和), *Member*, *CCF*, *ACM*, *IEEE*, and Yun-Ji Chen[1] (陈云霁), *Member*, *CCF*, *ACM*, *IEEE*

[1] *State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

E-mail: {chenpeng, zlei, yinhes, cyj}@ict.ac.cn

Received October 30, 2013; revised January 3, 2014.

Abstract The combination of growing transistor counts and limited power budget within a silicon die leads to the utilization wall problem (a.k.a. "Dark Silicon"), that is only a small fraction of chip can run at full speed during a period of time. Designing accelerators for specific applications or algorithms is considered to be one of the most promising approaches to improving energy-efficiency. However, most current design methods for accelerators are dedicated for certain applications or algorithms, which greatly constrains their applicability. In this paper, we propose a novel general-purpose many-accelerator architecture. Our contributions are two-fold. Firstly, we propose to cluster dataflow graphs (DFGs) of hotspot basic blocks (BBs) in applications. The DFG clusters are then used for accelerators design. This is because a DFG is the largest program unit which is not specific to a certain application. We analyze 17 benchmarks in SPEC CPU 2006, acquire over 300 DFGs hotspots by using LLVM compiler tool, and divide them into 15 clusters based on graph similarity. Secondly, we introduce a function instruction set architecture (FISC) and illustrate how DFG accelerators can be integrated with a processor core and how they can be used by applications. Our results show that the proposed DFG clustering and FISC design can speed up SPEC benchmarks 6.2X on average.

Keywords dataflow graph, many-accelerator, clustering, function instruction set architecture

1 Introduction

Transistor density and speed continue to increase with Moore's Law. However, the limitation of threshold voltage scaling is ushering an era of no-classical scaling and the power efficiency of devices is growing slowly[1]. Both growing transistor count and limited power budget cause the utilization wall phenomenon, a.k.a. Dark Silicon, that limits the fraction of chip which can run at full speed. For example, prior studies showed that with 45nm TSMC process, less than 7% of a 300 mm^2 processor die can run at full frequency with 80 W power budget[2]. This percentage will be less than 3.5% in 32 nm according to the ITRS roadmap projections and CMOS voltage scaling theory.

To fully utilize the abundant on-chip transistor resource within energy envelope, customizing hardware accelerators for specific application domains is an effective approach to tackling architecture design challenges such as "power wall", "utilization wall". This approach has great potential for energy-efficiency improvement, and attracts the attention of both academia and industry. Recent studies have shown 1 000∼10 000X efficiency improvement over mainstream processors[3-5].

Generally, there are two major issues that need to be solved for accelerator-based general-purpose processor architecture design, which is called many-accelerator architecture. First, we should decide what kind of and how many accelerators need to be integrated on chip to maximize the coverage of applications. Currently, most researches on accelerator design are coarse-grained and mainly focus on applications, e.g., network processor[6], multimedia processor[7], or algorithms, e.g., machine learning[8], encryption and decryption[9], with a narrow applicable range. Secondly, we need to solve the issue that when many accelerators are integrated on chip, how applications and programmers use them to improve energy-efficiency, i.e., programming model. Currently, in most of the accelerator-based systems, applications should be aware of the underlying hardware resource

Regular Paper

This paper is supported by the National Natural Science Foundation of China under Grant Nos. 601173006, 61221062, and the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA06010403.

and explicitly map their code to specific hardwired module. This "manual partition" is of a great burden for programmers.

Based on the above analysis, we achieve two-fold contributions in this paper. First, we propose a novel accelerator design method by clustering of dataflow graphs (DFGs) of basic blocks (BBs) in application hotspot. The hotspot code regions consume major execution time and are generally considered to be the target of accelerator design. The basic block refers to the code region which has single entry and exit, and is the largest program unit that is not specific to a certain application. The dataflow graph is a directed-acyclic graph (DAG) representing the operations and data dependencies within each basic block. By profiling SPEC CPU 2006 benchmarks using LLVM compiling tool, we acquire more than 300 hotspot DFGs and group them into several clusters based on graph similarities among them. We then design a hardware accelerator for each cluster. The principle behind our approach is to find the maximum intersection among general-purpose applications to promise both coverage and chip area efficiency.

Second, when we have a set of DFG accelerators, we propose to define a customized macro instruction, which we call "function instruction" for each accelerator. We implement the last pass of LLVM compiler backend to search specific DFG and replace it with one function instruction and its corresponding configurations on accelerators. When such function instructions are decoded in pipeline, the configuration data will be loaded to designated accelerator. Thus, from the processor's perspective, each accelerator is a functional unit, which is exactly similar to ordinary function units like adder, multiplier. Such Function Instruction Set Computer (FISC), which corresponds to CISC and RISC architecture, can make underlying hardware accelerators entirely transparent to programmers. FISC also eases the addition of more accelerators on chip.

To summarize, the main contributions we have made in this paper include:

• A new accelerator design method based on hotspot DFGs analysis and clustering. We define features for DFGs and propose to use graph similarity algorithm for DFG clustering.

• Function Instruction Set Computer (FISC) design method, which is efficient for many-accelerator architecture.

• A set of application profiling tools and backend code generation tools based on LLVM and clang framework.

The following of the paper is organized as follows. Section 2 introduces the approach of DFGs clustering. Section 3 describes the accelerator design for each DFG cluster. Section 4 introduces the FISC computer architecture and Section 5 presents its evaluation results. Section 6 discusses related work and Section 7 concludes the paper.

2 DFGs Clustering

In this section, we describe the DFGs clustering approach in details. First, we give a brief introduction for LLVM compiler tool for profiling. Second, we introduce similarity flooding to compute the similarity between DFGs. Finally, agglomerative clustering algorithm is adopted to classify the DFGs training set.

2.1 Application Profiling Using LLVM

LLVM, a compiler framework, can collect high-level information about arbitrary programs during compile-time, link-time, run-time, and support transparent, lifelong program analysis[10]. LLVM passes perform the transformations and optimizations for programs and become the most interesting part of a compiler. In order to get the dataflow graphs of basic blocks in the hotspot, we implement three passes.

1) Instruction count pass: records the execution time of every function.

2) Function processing pass: breaks the recursion between functions and selects the loops in the hot functions which have high weight of execution time.

3) Loop processing pass: the basic blocks of hot loops are processed and dataflow graphs are produced.

Fig.1 describes the profiling procedure with LLVM tool. Dataflow graphs are produced from original programs after seven processing steps.

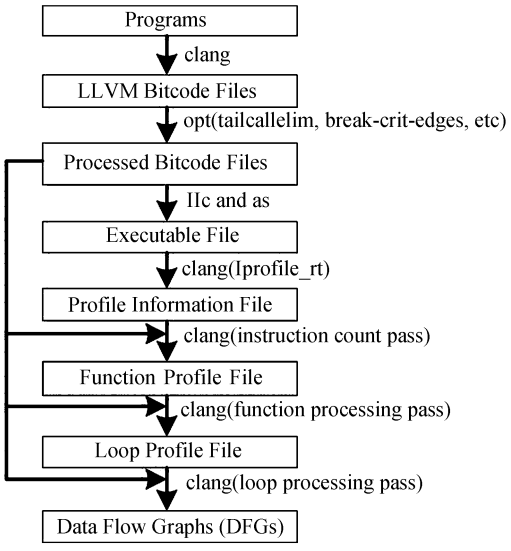


Fig.1. Profiling procedure with LLVM tool.

The benchmark we use in this paper is SPEC CPU 2006 suite which is intended to be diverse and an ideal

candidate for wide range workloads analysis. We modify clang, the C language family frontend of LLVM for our profiling purpose. However, clang only supports program languages including C, Objective-C, C++, and Object C++. As a result, we choose 17 benchmarks of SPEC CPU 2006, which are written with C/C++. SPEC CPU 2006 provides four kinds of input and output datasets for benchmarks — "all", "ref", "test", and "train". Table 1 lists their names and input sets of the benchmarks, where "test/input" refers to the input data of the "test" set, "all/input" refers to the input data of the "all" set.

Table 1. Benchmarks Chosen for Analysis and Their Input Sets

Name	Input Set	Name	Input Set
astar	Test/input	milc	Test/input
bzip2	Test/input	mcf	Test/input
dealII	All/input	namd	All/input
gcc	Test/input	perlbench	Test/input
gobmk	Test/input	povray	Test/input
h264ref	Test/input	sjeng	Test/input
hmmer	Test/input	soplex	Test/input
lbm	Test/input	sphinx3	Test/input
libquantum	Test/input		

After the profiling, we get over 300 dataflow graphs. They come from loops of hot functions and account for major program execution time. The instruction number of them varies from 5 to 50. Intuitively, we can design hardware accelerator for each hot DFG. However, it is cumbersome and unacceptable regarding the silicon estate efficiency. We propose to cluster these DFGs based on their similarities among each other to reduce the number of accelerators thus maximize their utilization. We use 200 randomly chosen DFGs for clustering (training set) and the remaining graphs are used for evaluation (testing set).

2.2 DFGs Similarity

Prior researches on program similarity analysis mainly use features like instruction mix, both static and dynamic, load distance, etc. And the goal of program analysis is optimization. However, in this work we do program analysis and clustering for accelerator design. Generally, the most high performance and energy-efficient design method of hardware for a piece of code is to match its dataflow. Thus we consider both graph similarity and other well-known program characteristics including dynamic instruction mix, load distance, memory footprint, and instruction/data level parallelism for basic blocks comparison and clustering.

For the graph comparison problem, a simple "yes" or "no" result can be given with the help of traditional algorithms. Instead, we adopt a versatile graph matching algorithm, similarity flooding, to compute the approximate similarity between two graphs[11]. Mapping and scoring are two most important parts of similarity computing between two graphs. During the first stage, we will pair vertex in graph A with certain vertex in graph B, which is the most similar to the former one, based on the instruction type of vertex and neighborhood information. After the mapping for all vertexes, we assign a similarity score for two input graphs.

Fig.2 shows a simple example of similarity flooding algorithm used in this paper. A and B are the simplified DFGs that need evaluation. The vertexes of them stand for the instructions in basic blocks and different shapes indicate the instruction types. Edges between nodes describe the data dependence of them. Detailed steps in similarity flooding will be explained in the following subsections.

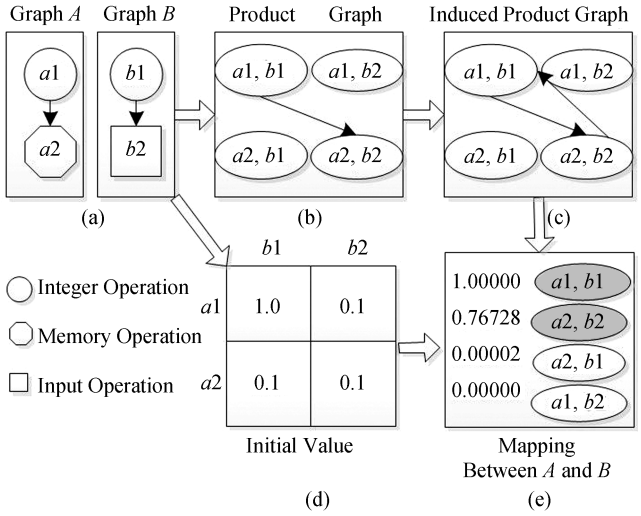


Fig.2. Simple example to illustrate the similarity flooding algorithm.

2.2.1 Product Graph Construction

A product graph contains all the possible map pairs of vertexes and edges derived from its two input graphs. All the later computations about similarity are based on the product graph. In order to define the construction of it, we annotate graphs A and B with (V_A, E_A) and (V_B, E_B). The vertex set (V_P) and edge set (E_P) in the product graph are defined as follows:

$$V_P = \{M_x N; M \in V_A, N \in V_B\}, \tag{1}$$

$$E_P = \{(M_x N) \times (M' \times N'); M \times M' \in E_A, \; N \times N' \in E_B\}. \tag{2}$$

Each vertex in the product graph is from $A \times B$. According to similarity flooding algorithm, the intu-

ition behind edges such as $((a1, b1), (a2, b2))$ is that if $a1$ is similar to $b1$, the similarity between $a2$ and $b2$ is probable higher. The value of vertex in the product graph stands for the similarity between corresponding vertexes in the two original graphs.

2.2.2 Induced Product Graph

As above mentioned, the similarity of a vertex in the product graph will propagate for its neighbors along edges. However, the end vertex of an edge should make its similarity affect the start vertex. So, we need to add an edge in the opposite direction for every edge in the original product graph. Just as Fig.2 mentioned, the edge from $(a2, b2)$ to $(a1, b1)$ is added. Now the weights will be placed on the edges and indicate how well the similarity of the start vertex affects its neighbors, ranging from 0 to 1. The weight is assigned as follows:

$$W((M \times N) \times (M' \times N')) = \frac{1}{|\{(M \times N) \times (J \times K), J \times K \in V_P\}|}, \quad (3)$$

where (J, K) denotes the vertex connected with (M, N). In other words, the weight of an edge equals to the reciprocal of the out-degree of (M, N). The assignment is based on the intuition that every edge starting from (M, N) makes equal contribution to spreading the similarity of (M, N) to its neighbors.

2.2.3 Iteration

Before fixed number of iterations, we should assign initial value for each vertex in the product graph. It is based on the shapes of two corresponding vertexes in the original graph. In Fig.2, because of the same operation type of $a1$ and $b1$, vertex $(a1, b1)$ in the product graph is given the initial similarity value of 1.0.

The similarity value of each vertex needs to be propagated along edges. Let $\sigma^k(M \times N)$ be the similarity value of node $(M, N) \in V_P$ after k iterations and $\sigma^0(M \times N)$ equals the initial value of (M, N). According to the similarity flooding, $\sigma^k(M \times N)$ is computed with the values of itself and its neighbors:

$$\sigma^k(M \times N) = \text{normalize}(\sigma^{k-1}(M \times N) + \sigma^0(M \times N) + \varphi(\sigma^{k-1}(J \times K) + \sigma^0(J \times K))). \quad (4)$$

The function φ increases the similarity of each vertex in the induced product graph based on similarities of its neighbors. After each iteration, normalization should be done and make $\sigma^k(M \times N)$ range from 0 to 1.

The iteration will continue until the Euclidean length of the residual vector $\mathbf{\Delta}(\sigma^n, \sigma^{n-1})$ becomes less than ε[5]. To solve the possible misconvergence, the maximal iteration number will be used. After the iterations, the similarity values of all vertexes in the induced product graph become stable. We sort these values in the decreasing order, as Fig.2(e) describes. In order to get the maximal cumulative similarity, vertexes with higher values are chosen to measure the similarity of two original graphs. Then we get the vertexes set M_P containing the matching result. In our example, $M_P = \{(a1, b1), (a2, b2)\}$.

As a result, we define the scores between two input graphs as follows:

$$SCORE(A, B) = \frac{\sum_{(J,K) \in M_P} (\sigma^0(J, K))}{\max\{|V_A|, |V_B|\}}. \quad (5)$$

After quantitative comparisons for 200 dataflow graphs, a 200×200 matrix named ***SCORE*** is produced. Its element, $SCORE[i][j]$, measures the similarity between graph i and graph j.

2.3 DFG Clustering and Analysis

Now that we have the similarity matrix, the next step is to divide 200 training cases into several clusters. In some famous clustering algorithms such as agglomerative clustering, k-means, or SOM, elements which have the highest similarity will be taken together. Agglomerative clustering is a "bottom up" method of hierarchical clustering. The pair of clusters which has the highest similarity will be merged as one during iterations. If we record the process of clustering, we will get different clustering results. The number of clusters will vary from N (the number of elements being processed) in the beginning to 1 in the end.

Fig.3 is an example showing the agglomerative clustering. Six dataflow graphs ($a \sim f$) are shown here. The element of 6×6 matrix, $SCORE[i][j]$, refers to the similarity between graph i and graph j. Five iterations need to be done during the execution of clustering. In the beginning, each cluster has a graph and the number of clusters is 6. After an iteration, the most similar graphs, b and c, are merged into one cluster and the cluster number decreases to 5. The similarity values within clusters and the cluster size will vary along with iterations.

To evaluate the quality of clustering, we define the correlation between clusters and within a cluster. Firstly, we define the diversity of two clusters M_1 and M_2 as follows:

$$Diversity(M_1, M_2) = 1 - \frac{\sum_{a \in M_1} \sum_{b \in M_2} Score[a][b]}{|M_1||M_2|}. \quad (6)$$

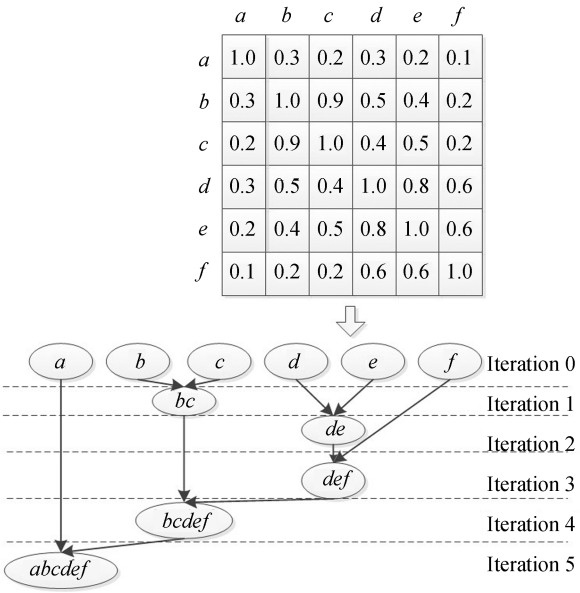


Fig.3. Simple example illustrating the agglomerative clustering.

Higher diversity means more disparity between clusters, or less overlap between clusters. Meanwhile, we define the convergence within a cluster as:

$$Convergence(M_1) = \frac{\sum_{a \in M_1} \sum_{b \in M_1} Score[a][b]}{|M_1|^2}. \quad (7)$$

Convergence describes how individual DFGs within a cluster close to each other. For a more converged cluster, it will be much easier for hardware design to abstract features.

Fig.4 shows the diversity and convergence of clusters when the number of clusters decreases. It can be seen that clusters themselves become less converged while more diverged with others as the number of clusters decreases. When the cluster number becomes 15, the diversity among clusters increases to 0.437 and the convergence within clusters still keeps high at 0.884. Thus, in the following of this paper, we choose 15 clusters for evaluation.

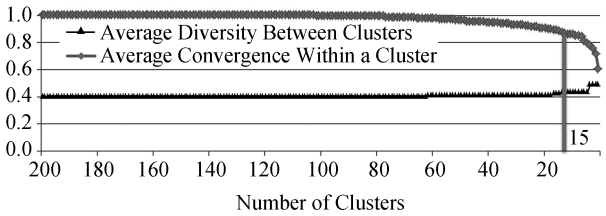


Fig.4. Average diversity between clusters and convergence with a cluster.

Fig.5 describes the number of dataflow graphs in each cluster and the benchmarks covered by each cluster. The first observation is that different application programs do have similar DFGs, which fits our common expectation, because different applications have similar hotspot piece of code. Second, as Fig.5 shows, 12 of the clusters cover more than 4 programs while 2 of them cover 10 programs. This demonstrates the applicability of our clusters.

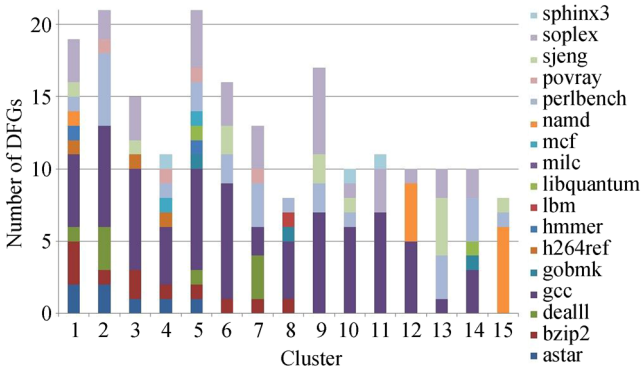


Fig.5. Number of dataflow graphs in each cluster and coverage of SPEC benchmarks.

3 Designing Accelerators for Each DFG Cluster

Now that we have clusters containing many DFGs, the next issue is how to design a hardware accelerator for each cluster to maximally fit all DFGs in it. We propose to find one representative DFG within a cluster and design fixed hardwired module for it. The representative DFG is the one with the closest distance with all the others in terms of average similarity between two DFGs. Designing hardwired module for it can promise the match to all DFGs in the cluster to the maximum extent. Lastly, we use reconfigurable functional unit and interconnection to match other parts for individual DFGs.

To evaluate the effectiveness of the 15 selected DFGs, we use 100 DFGs from SPEC benchmarks as testing set as mentioned in Subsection 2.1. We evaluate the similarity between 100 testing DFGs and the representative DFGs of the cluster to decide which cluster the testing DFG belongs to. The results are shown in Fig.6. It can be seen that 90% of the testing DFGs

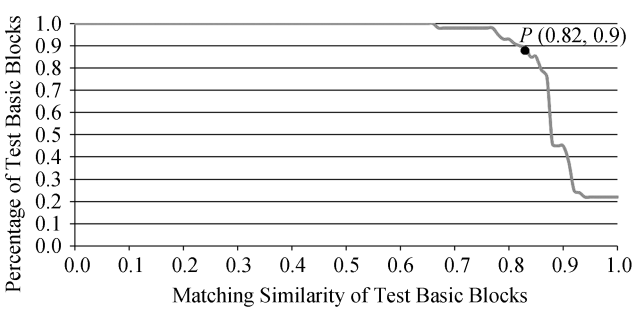


Fig.6. Coverage of clusters for 100 testing DFGs from SPEC.

have higher than 82% similarity value with our clusters as Fig.6 shows. This proves the good coverage and applicability for our clusters.

Fig.7(a) is one of the 15 representative DFGs as mentioned above. Fig.7(b) describes a pipelined accelerator design for it. Fig.7(a) has four types of vertexes: square, circle, double circle, and octagon. The square refers to the data dependence with neighbor basic blocks. The data produced by neighbor basic blocks will be transferred to the current block. The circle stands for integer operation and the double circle for float operation. The octagon describes the memory operation.

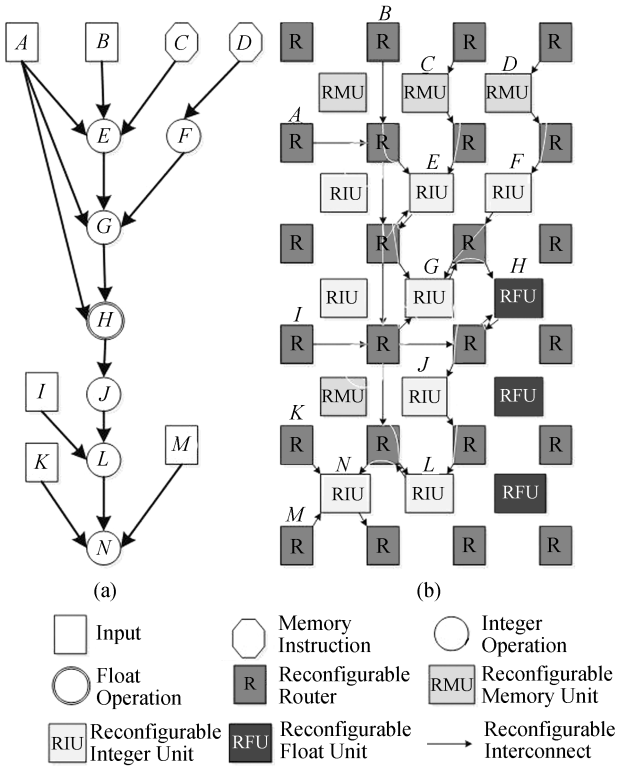


Fig.7. Accelerator design for DFGs.

Accordingly, in the accelerator design, four types of vertexes in the dataflow graph will be mapped to different kinds of reconfigurable units in Fig.7(b): reconfigurable router, reconfigurable integer unit, reconfigurable floating point unit, and reconfigurable memory unit to load/store data from/to memory. For example, vertex E in the dataflow graph will be mapped on a reconfigurable integer unit and make integer operation. By mapping the hotspot DFGs onto the accelerator, we can achieve much better performance and energy-efficiency, because many energy-hungry operations such as instruction fetch, decode, data movement will be eliminated.

4 Function Instruction Set Computer Architecture

In above sections, we describe our approaches on the problem of what kind of and how many accelerators need to be integrated on chip to maximize the coverage of applications. In this section, we introduce the Function Instruction Set Computer (FISC) architecture, to tackle the problem of the integration of accelerators with processor cores and its corresponding programming model.

The simplified architecture of FISC is shown in Fig.8. In FISC we introduce 15 macro instructions, called function instructions, each of which corresponds to a DFG accelerator. As shown in Fig.8, at the decode stage of FISC pipeline, the processor core can identify an instruction to be a regular or a function instruction. Regular instructions follow traditional pipelines, while a function instruction will load configurations into a dedicated accelerator as indicated by the instruction. The accelerators as described above can access register file and memory hierarchy of the processor core. Regular instructions will not stall unless they have data dependency on function instructions. The register file is the place where regular and function instructions exchange data.

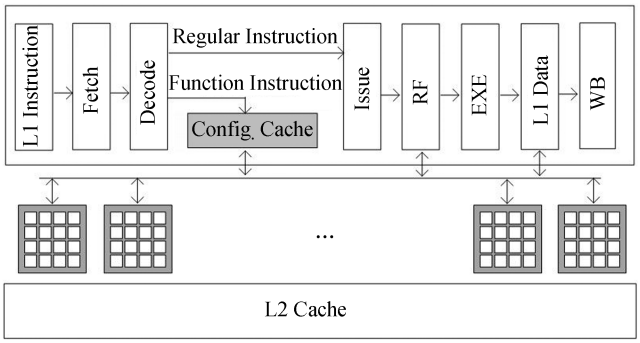


Fig.8. FISC: general-purpose many-accelerator architecture.

We have implemented the last pass of LLVM compiler tool to annotate the DFGs that can be mapped to a certain accelerator within the application code, and replace the basic block code with one function instruction and its corresponding reconfiguration information. By doing this, FISC is totally compatible with current programming model and the underlying hardware is entirely transparent to programmers. This will greatly ease the "manual partition" burden of prior accelerator-based system design.

5 Evaluation

In this section, we show the evaluation results of our proposed FISC architecture. In SimpleScalar tool set

is modified to simulate real programs running on FISC architecture. We choose ARM as the basic instruction set and add 15 macro function instructions.

Firstly, in Fig.9, we show the breakdown of the execution time of benchmarks on FISC architecture. It is clear that the applications have spent most of their execution time on accelerators. This is because our accelerators are designed for hotspot code regions, which occupy much of the execution time of a program. The results again prove the good coverage of our clusters.

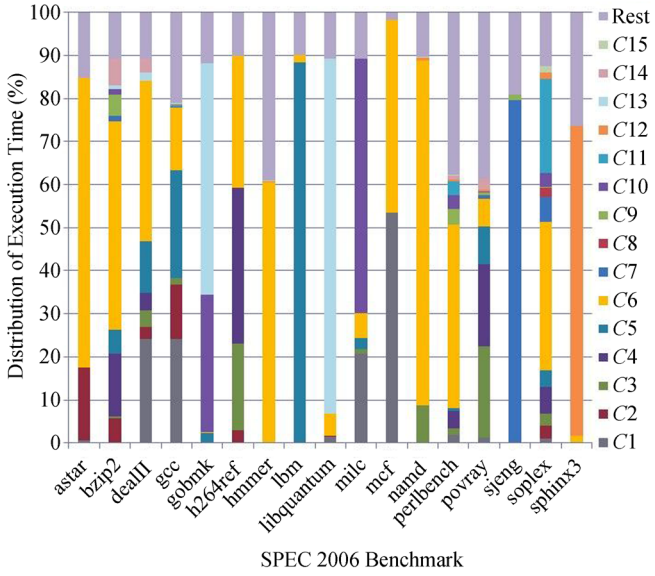


Fig.9. Breakdown of benchmarks' execution time.

Finally, Fig.10 shows the simulation results of speedup for all selected benchmarks. We can achieve 6.2X on average speedup when compared with the baseline model. The baseline is the ARM processor as shown in Fig.10 without accelerators integrated.

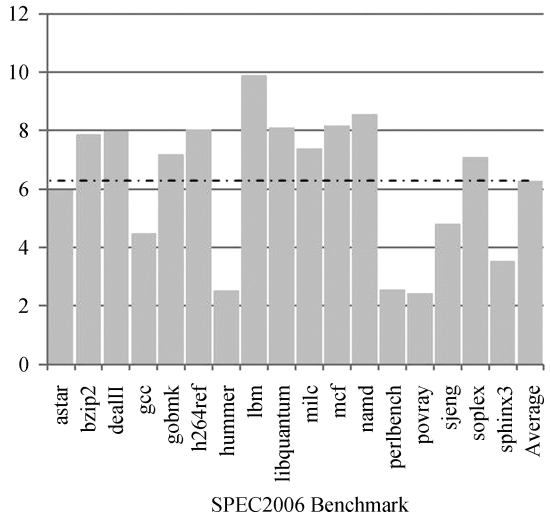


Fig.10. Speedup for each benchmark on FISC.

6 Related Work

The related research topics include workload characterization for hardware and software optimization efficiency. A great number of specialized processors exist for applications and domains such as encryption and decryption[12], streaming multimedia processing[13-14], vector streaming processing[15], physical simulation[16] and computer graphics[17-19]. Although great energy-efficiency is achieved, they cannot be capable for various workloads. The 10×10 processor[3] is designed for general-purpose workload space. It is a multi-core architecture and each engine is specialized for different workload groups. However, code region uncovered by multiple cores will be difficult to deal with under the architecture without a general-purpose processor core. In addition, its clustering method is still based on the instruction mix and the result of clustering has little value in engine design. CENTRIFUGE[20] is used for graph clustering for program characterization. But it does not mention basic block level workload clustering method and the detailed hardware design. Behavior-level observability analysis[21] was proposed for low power design. However, the concept is mainly used for RTL synthesis tools and does not describe the workloads analysis.

7 Conclusions

With the motivation of solving the contradiction between utility and energy efficiency in the accelerator's design, we took detailed analysis for general-purpose workloads. Seventeen programs of SPEC CPU 2006 were selected for our LLVM analysis tool and hundreds of dataflow graphs were produced. Then we adopted similarity flooding algorithm and agglomerative clustering method to divide the graphs into 15 clusters. The clustering result keeps low similarity between clusters and high similarity within the cluster. We selected 15 representative graphs which have high average similarity with others in the same cluster and tested their coverage by testing basic block graphs. We found that at least 90% of test graphs can be covered by representations. Finally, 15 pipelined accelerators were designed for each representative graph. Programs from SPEC CPU 2006 can achieve as much as 6.2X average speedup with little extra power consumption. Therefore, our basic block level dataflow graph analysis can improve the applicability of hardware accelerator and relieve the contradiction between energy-efficiency and applicability.

References

[1] Govindaraju V, Ho C H, Sankaralingam K. Dynamically spe-

cialized datapaths for energy efficient computing. In *Proc. the 17th Symp. High Performance Computer Architecture* (*HPCA*), February 2011, pp.503-514.

[2] Venkatesh G, Sampson J, Goulding N *et al.* Conservation cores: Reducing the energy of mature computations. *ACM SIGARCH Computer Architecture News*, 2010, 38(1): 205-218.

[3] Guha A, Zhang Y, ur Rasool R *et al.* Systematic evaluation of workload clustering for extremely energy-efficient architectures. *ACM SIGARCH Computer Architecture News*, 2013, 41(2): 22-29.

[4] Cong J, Ghodrat M A, Gill M *et al.* Architecture support in accelerator-rich CMPs. In*Proc. the 49th Annual Design Automation Conference* (*DAC*), June 2012, pp.843-849.

[5] Hameed R, Qadeer W, Wachs M *et al.* Understanding sources of inefficiency in general-purpose chips. In *Proc. the 37th ISCA*, June 2010, pp.37-47.

[6] Memik G, Memik S O, Mangione-Smith W H. Design and analysis of a layer seven network processor accelerator using reconfigurable logic. In *Proc. the 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002, pp.131-140.

[7] Yoon C W, Woo R, Kook J *et al.* An 80/20-MHz 160-mW multimedia processor integrated with embedded DRAM, MPEG-4 accelerator and 3-D rendering engine for mobile applications. *IEEE Journal of Solid-State Circuits*, 2001, 36(11): 1758-1767.

[8] Steinkraus D, Buck I, Simard P Y. Using GPUs for machine learning algorithms. In *Proc. the 8th Int. Conf. Document Analysis and Recognition*, August 29-September 1, 2005, pp.1115-1119.

[9] Pionteck T, Staake T, Stiefmeier T *et al.* Design of a reconfigurable AES encryption/decryption engine for mobile terminals. In *Proc. Int. Symp. Circuits and Systems*, May 2004, Vol.2, pp.545-548.

[10] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, March 2004, pp.75-86.

[11] Melnik S, Garcia-Molina H, Rahm E. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proc. the 18th Int. Conf. Data Engineering*, March 2002, pp.117-128.

[12] Wu L, Weaver C, Austin T. CryptoManiac: A fast flexible architecture for secure communication. In *Proc. Int. Symp. Computer Architecture*, June 30-July 4, 2001, pp.110-119.

[13] Ebeling C, Cronquist D C, Franklin P. RaPiD — Reconfigurable pipelined datapath. In *Proc. the 6th International Workshop on Field-Programmable Logic*, Sept. 1996, pp.126-135.

[14] Goldstein S C, Schmit H, Moe M *et al.* PipeRench: A coprocessor for streaming multimedia acceleration. In *Proc. the 26th Int. Symp. Computer Architecture*, May 1999, pp.28-39.

[15] Ahn J H, Dally W J, Khailany B *et al.* Evaluating the imagine stream architecture. In *Proc. the 31st Int. Symp. Computer Architecture*, June 2004.

[16] Boeing A, Braunl T. Evaluation of real-time physics simulation systems. In *Proc. the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, December 2007, pp.281-288.

[17] Luo Z, Liu H, Wu X. Artificial neural network computation on graphic process unit. In *Proc. Int. Joint Conf. Neural Networks*, July 31-Aug. 4, 2005, Vol.1, pp.622-626.

[18] Lindholm E, Nickolls J, Oberman S *et al.* NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 2008, 28(2): 39-55.

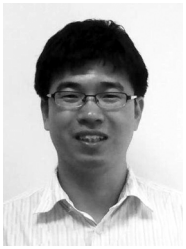
[19] Owens J D, Luebke D, Govindaraju N *et al.* A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 2007, 26(1): 80-113.

[20] Demme J, Sethumadhavan S. Approximate graph clustering for program characterization. *ACM Transactions on Architecture and Code Optimization* (*TACO*), 2012, 8(4): Article No. 21.

[21] Cong J, Liu B, Majumdar R *et al.* Behavior-level observability analysis for operation gating in low-power behavioral synthesis. *ACM Transactions on Design Automation of Electronic Systems* (*TODAES*), 2010, 16(1): Article No.4.

Peng Chen received his B.S. degree from the Department of Computer Science and Technology, Hefei University of Technology, in 2012. He is currently a master's student in the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. His research interests include program analysis and accelerator design.

For the technical biography of **Lei Zhang**, please refer to p.238.



Yin-He Han received the M.S. and Ph.D. degrees in computer science from ICT, CAS, in 2003 and 2006, respectively. He is currently an associate professor at ICT, CAS, His research interests include VLSI/SOC interconnection, testing and fault-tolerance. Dr. Han is a recipient of the Best Paper Award at Asian Test Symposium 2003. He is a member of CCF, ACM, IEEE society. He was the program co-chair of Workshop of RTL and High Level Testing (WRTLT) in 2009, and serves on the Technical Program Committees of several IEEE and ACM conferences, including ATS, GVLSI, etc.

Yun-Ji Chen graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, in 2002. Then, he received the Ph.D. degree in computer science from ICT, CAS, in 2007. He is currently a professor at ICT. His research interests include parallel computing, microarchitecture, hardware verification, and computational intelligence. He has authored or coauthored one book and more than 40 papers in these areas.