STRAIGHT: Realizing a Lightweight Large Instruction Window by using Eventually Consistent Distributed Registers

Hidetsugu IRIE*, Daisuke FUJIWARA*, Kazuki MAJIMA*, Tsutomu YOSHINAGA*
*The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan
E-mail: irie@is.uec.ac.jp, Dz-Fujiwara@comp.is.uec.ac.jp, majima@comp.is.uec.ac.jp, yosinaga@is.uec.ac.jp

***Abstract*—As the number of cores as well as the network size in a processor chip increases, the performance of each core is more critical for the improvement of the total chip performance. However, to improve the total chip performance, the performance per power or per unit area must be improved, making it difficult to adopt a conventional approach of superscalar extension. In this paper, we explore a new core structure that is suitable for manycore processors. We revisit prior studies of new instruction-level (ILP) and thread-level parallelism (TLP) architectures and propose our novel STRAIGHT processor architecture. By introducing the scheme of distributed key-value-store to the register file of clustered microarchitectures, STRAIGHT directly executes the operation with large logical registers, which are written only once. By discussing the processor structure, microarchitecture, and code model, we show that STRAIGHT realizes both large instruction window and lightweight rapid execution, while suppressing the hardware and energy cost. Preliminary estimation results are promising, and show that STRAIGHT improves the single thread performance by about 30%, which is the geometric mean of the SPEC CPU 2006 benchmark suite, without significantly increasing the power and area budget.**

I. Introduction

With the greater use of multicore/manycore structures, the total chip performance is determined by the balance of the core count and performance of each core [1]. Consequently, it is more effective to reduce the power or transistors per instruction operation than to enhance a core by using aggressive speculation or sophisticated scheduling. Conventional approaches that enhance the single thread performance by exploiting increased transistor count (which is enabled by Moore's law) are no longer attractive.

Conversely, the single thread performance per watt or per unit area becomes more significant. As the core count increases, the latency of the single thread part severely affects the entire performance. Moreover, the improvement of the core architecture results in an improvement of all cores in a processor, which operates a scale-out application in parallel. These effects are not easily realized by improving the multicore structure or interconnection networks.

This paper introduces the STRAIGHT core architecture that effectively improves the single thread performance with a lightweight mechanism. It is inspired by a distributed key-value store scheme that is used in the areas of scale-out cloud computing. STRAIGHT is provided with a distributed register file structure that is free from centralized management, including structures for allocation and invalidation. Register renaming is eliminated and reordering is performed in an eventually consistent manner, which significantly reduces the control hardware. This feature enables a scalable extension of the instruction window and subsequently, the exploitation of instruction level parallelism (ILP) from larger sections of the running program. Alternatively, schedulers and data paths are kept small to increase the speed and efficiency of the implementation, which still has sufficient bandwidth to execute ILP programs or non-scale-out thread level parallelism (TLP) programs. For scale-out applications, we assume the manycore processor structure, which consists of a number of STRAIGHT architecture cores (SAC) that are loosely connected each other.

Being the first report on this novel processor architecture, in this paper, we discuss the concept behind STRAIGHT, propose basic principles, and estimate the performance and budget expectation. The rest of the paper consists of following sections. Section II revisits studies of new architectures that were designed to improve the ILP/TLP performance of superscalar processors, and discusses the dilemma of both scalability approach and quick worker approach. In section III, we discuss the key idea of STRAIGHT that allows the resolution of this dilemma by introducing a distributed key-value store to the processor architecture. Software and hardware outline models of STRAIGHT are described in section IV. Section V estimates the performance of STRAIGHT by using a cycle-accurate superscalar simulator and possible parameters, as well as hardware budgets. Finally, we summarize the paper in section VI.

II. Previous Studies Based on new ILP/TLP Architectures

A. Pursuit of Scalability

Generally, programs are known to contain a large amount of instruction-level parallelism that is yet to be exploited [2]. To exploit this parallelism, a large-scale single thread core that is provided with large instruction window and large issue width is required, along with optimized memory disambiguation. In such an approach where the issue width is extended, the centralized structure of the "critical loop" in the superscalar architecture becomes a major bottleneck [3]. To execute dependent instructions in the consecutive cycles, the circuit delay of select-to-wakeup loop in the scheduler and result forwarding loop in data path must be in a cycle period. However , the complexity of those loops exponentially increases when the issue width is increased in the superscalar architecture because the entire instruction window entries or functional units are completely connected. To address this problem, several studies were performed that prunes this connection by adopting clustered structures that partition and distribute processor components, such as multiscalar processors [4], PEWs [5], multicluster [6], complexity-effective superscalar [3], cost-effective clustered architecture [7], and ILDP [8]. DEC Alpha 21264 also adopted a two-node clustered structure for the datapath [9]. Additional communication latency is required when inter-node communication occurs in these clustered microarchitectures. Thus, the use of sophisticated control algorithms that localize a major portion of communications into intra node is a key technique. In particular, steering algorithms that assign nodes to each instruction in order to optimize load balance and localization [10]–[15] are important.

To achieve further scalability against wire delays, tiled architectures, which map many smaller nodes to physically

978-0-7695-4893-7/12 $26.00 © 2012 IEEE
DOI 10.1109/ICNC.2012.66

compatible networks such as 2D-mesh [16], [17], are introduced. Small nodes are operated with rapid clock frequencies and cooperate with each other. The dataflow of the program portion is directly mapped to tiled execution units and the portion is executed by the entire chip. To also consider networks for caches, NUCA is proposed, which has a variable hit latency that is determined by the physical location of the core and cache [18].

B. Utilization of Redundant Pipelines

The issue width extension has been enabled by such clustered structures. However, they need sufficient instructions to be executed in parallel. It is relatively hard to make the instruction window scalable, even in a clustered structure, because it requires coherent register management. Thus, it is usually difficult for large-scale clustered microarchitectures to maintain pipeline usage because they have to exploit ILP from relatively small instruction windows. SMT is a technique that can be used to fill the pipeline by executing instructions of other threads with redundant resources [19], and is therefore effective for scalable architectures [20]. Speculative multithreading approaches, which exploit the multi-thread mechanism to increase single thread performance, are also suitable for scalable architectures as well as chip-multi-processors. In speculative multithreading, single threads are speculatively divided into multiple threads and are executed in parallel [21], [22]. In particular, the slipstream processor creates helper threads from a single thread program and simultaneously runs both helper and main thread [23]. The program code of the helper threads is pruned; hence, its execution may be incorrect and the results are discarded. However, the helper thread can precede the main thread and its execution has the effect of preparing predictors and caches. However, for today's multicore processors, these approaches that operate speculative workloads by using redundant resources are required to achieve a sufficient improvement in performance to match their additional power consumption.

C. Quick Worker Approach

The other approach that is used to improve the single thread performance is reducing the latency of each instruction. Instead of operating many pipelines with complex control algorithms, this approach operates lightweight hardware that improves the number of instructions per cycle (IPC) by reducing the turnaround time for branch decisions, as well as the clock frequency. By eliminating complex resource controls and redundant pipelines, this approach is also desirable for power saving and available core counts. Examples of this approach are RTC [24] and Twin Tail [25]. For most cases, they achieve shorter pipeline; the former uses a trace cache to omit the rename stage, whereas the latter uses ALU networks to omit the issue stage. Thus, they achieve quicker instruction execution than conventional small superscalar.

D. Lessons from Recent Architectures

As previously discussed, to improve the core performance, there are two approaches that exploiting ILP on scalable structures, and lightweight pipelines with smart omitting. However, as a core element of future manycore processors, both of them suffer from a shortcoming. Although clustered or tiled architectures achieve a wide and flexible issue width that can operate at an efficient size for the running programs, parallelism in single thread programs or non-scale-out multi-thread programs is not sufficient to fill up the provided pipelines and interconnects. This tendency is triggered by the difficulty associated with achieving scalable instruction window. Several techniques were proposed that fill up pipelines with speculative execution; however, they are not attractive for current multicore processors because the performance efficiency of the entire chip may be reduced. Inherently distributed structures worsen the turnaround time of instruction execution because they have to manage instruction-node mapping and transfer register inquiries between nodes. The controls and data have to travel across wide areas of the chip for an instruction to be executed; however, in simple architecture, this can be completed in a compact area.

In contrast, the ability to realize performance improvement by using the lightweight approach is limited. The significant factor is the critical loop where clustered approaches also achieve higher clock latency. In addition, a reduction in other pipeline stages is not as significant. Although it still has the advantage of reducing the branch misprediction penalty. Moreover, the inevitable memory latency severely degrades the performance of this approach. In addition, smart omitting is achieved using a sophisticated centralized algorithm; hence it lacks the flexibility or scalability that are desirable for flexible manycore usage.

Consequently, improving the single thread performance suffers from the dilemma that it fails to match the power and hardware cost unless it exploits additional ILP; however, the use of a scalable structure for a large instruction window worsens the cost efficiency of an instruction execution because it requires additional control. Thus, studies of current multicore processors generally focus on the network or system structure design, as opposed to improving the core itself [1]. However, the significance of the performance per power or per unit area of the core increases as the core count increases with the processing rules. We believe that there is a different core design that is more suitable for the manycore era.

III. Concept of STRAIGHT Architecture

For each processor core that constitutes a manycore chip, an important feature is the performance or efficiency with which it executes a single thread or non-scale-out multi-thread program. It is not necessary to provide scalable wide pipelines for a core because the required maximum issue width is not very large for such applications. Although scalable wide pipeline structures were pursued in conventional clustered microarchitectures, presently, as a processing element of manycore processors, an adequate number of efficient pipelines is sufficient for a core. On the other hand, it is important to provide a larger instruction window for aggressive ILP execution. The effect of a large instruction window increases nowadays because prefetching has been improved, and the memory bandwidth is also improved by on-chip memory controllers and 3D stacking. Therefore, if a larger instruction window is realized, (the adequate number of) pipelines are filled by ILP execution without aggressive speculation. However, it is difficult to provide a large instruction window in conventional clustered microarchitectures because register management becomes a complex bottleneck.

Here we consider introducing a distributed key-value store scheme to a clustered microarchitecture, which can lead to agreement of the register file distribution between independent nodes without centralized mapping tables or broadcasting inquiries. For example, the node that holds the value of a certain register can be determined by hashing the register number into an m-bit modulo space, which is associated with n nodes such as Chord [26](Figure1). In this manner, neither the centralized register management nor broadcast for the register look-up that discourages the scalability of conventional clustered microarchitecture is eliminated. Thus enable us to create a number of instructions in-flight with a distributed scalable register file. Each node is responsible for an assigned register and the values are held in a cache-like associative table. The access latency of the distributed register file will increase; however, the reduced performance caused by this latency is known to small if the result bypass is adequately operated [27].

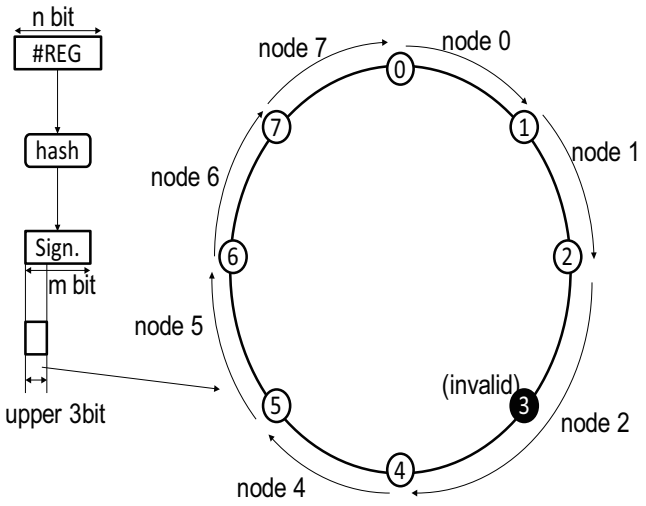

Fig. 1. Determination of the Responsible Node that Holds the Value of a Register

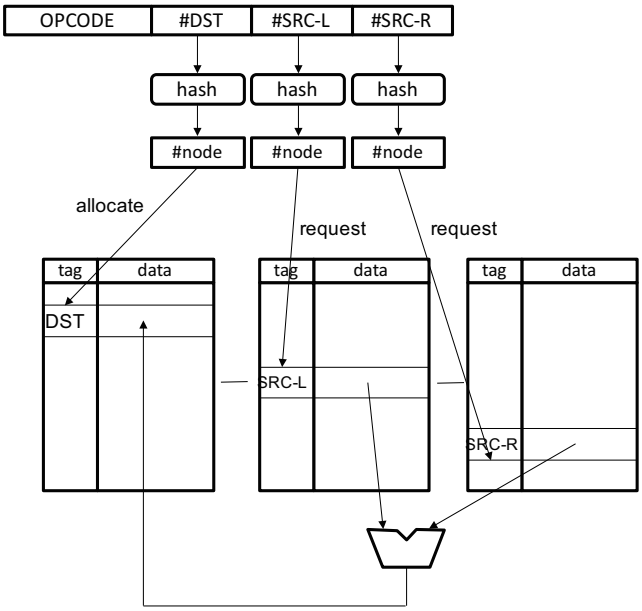

Fig. 2. Straight Execution on Distributed Register File

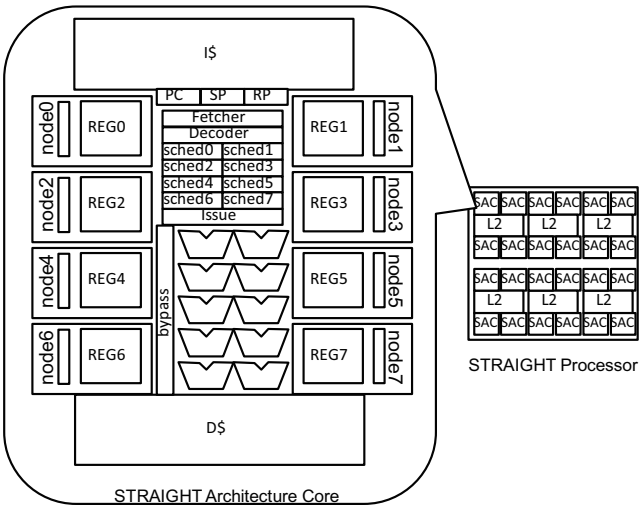

Fig. 3. Block Diagram of STRAIGHT architecture

To completely eliminate the centralized management of the register file, we have to solve register renaming and free list management as well as look-up. Here we introduce a supporting compiler that generates the code by using a huge logical space of write-once registers. For such a code, the processor enables to straightly hash the register number, determines the node, and performs the execution with distributed management. For example in Figure 2, the corresponding node of source and destination registers are directly determined from the instruction code by hashing each register field independently. Then, to execute the code, the register read requests are sent to the nodes that hold source L and R value respectively, and the value is attained by accessing the key-value table of that node locally. Similarly, the register allocation request is sent to the node that is responsible for the destination register, and an entry is allocated in the table. The execution is free from false dependency (Write-After-Write and Write-After-Read) because all of the registers are written once; therefore, the instruction can be executed as soon as both source register values are ready, in out-of-order manner. The allocated registers are implicitly freed when a certain number of subsequent instructions are fetched, which also enables us to achieve distributed management. This scheme is feasible because thousands of registers are available when such distributed management is available (a detailed feasibility study of code generation and hardware costs are described in Section IV). The assigned register entry is eventually invalidated after the last use and a new register is allocated to the entry. This approach will surpass conventional superscalar architecture by ILP execution from a large instruction window, reducing the spill memory access by large logical register files. Moreover, it does not require the cost for register renaming, and requires simpler branch managements, as described in Section IV.

Figure 3 shows the outline diagram of our STRAIGHT architecture. Although the register file is distributed, execution pipelines are centralized. The critical loop of issues and bypasses are centralized and the movement of critical data is kept within a small area (Figure 4). STRAIGHT exploits ILP from a larger instruction window that is enabled by the large distributed register file, directly executes the code that has a large logical register space, and has an efficient centralized conventional scale of execution pipelines. Using this STRAIGHT architecture core (SAC) as a basic component, the STRAIGHT processor consists of many SACs that are sparsely connected to each other. Many scale-out applications such as "map" and "reduce" can be executed in massively parallel by cooperation of many SACs in the STRAIGHT processor. In the next section, we describe the outline model of STRAIGHT from the software and hardware perspective including the instruction format, control model, and microarchitecture.

IV. Outline Model of STRAIGHT

A. Instruction Format

STRAIGHT has thousands of logical registers that can be used as general purpose registers; however they are written only once for lifetime. Program counters (PC), stack pointers (SP), and register pointers (RP, described later) are also provided as architectural registers, and they are rewritable. Instruction opcodes are similar to RISC architectures; however

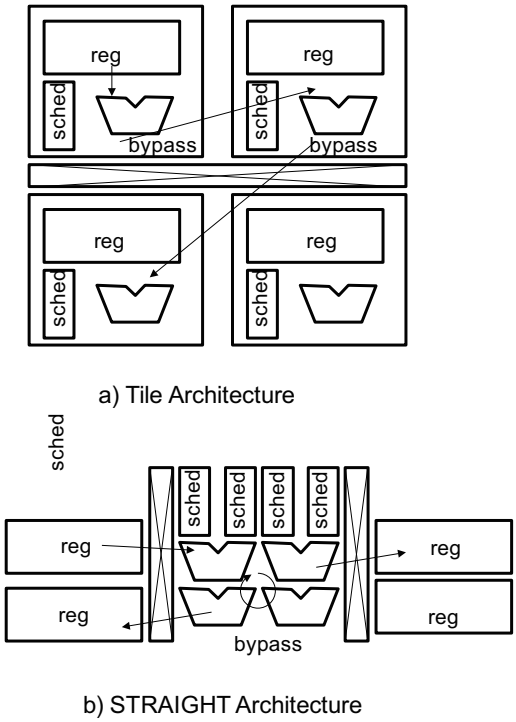

Fig. 4. Tile vs. STRAIGHT

```
s = a + b + c + d;

LD sp(0)          #ld a
LD sp(8)          #ld b
LD sp(16)         #ld c
LD sp(24)         #ld d
ADD [-4] [-3]     # s = a + b
ADD [-1] [-3]     # s = s + c
ADD [-1] [-3]     # s = s + d
ST [-1] sp(32)    #st s
```

Fig. 5. Source Registers in STRAIGHT

their operand manner is different. The value of RP at the decode time is used to indicate the destination register number (Figure 6). The RP value is incremented for every instruction decode operation. As a result, consecutive instructions have consecutive destination register numbers. The number of source registers is indicated in the source field L and R of the instruction format, which is given as the displacement from RP (Figure 5). The number of source registers is obtained by subtracting the displacement from RP of the decode time. The displacement has an upper bound; as a result, the register number, which is more distant from RP than the upper bound, is no longer accessed and can therefore be invalidated. RP is increased in a sufficiently larger modulo space than the upper bound of the displacement. The register number is reused after a sufficiently long invalidation period. If the value must be maintained beyond the upper bound, a "register move" or "store" instruction is inserted into the code.

Such an indication method is known to require an adjustment for the case of control flow join. In the STRAIGHT compiler that generates the code, both paths of the joining control flow move the corresponding registers in the same order. In addition, a special instruction that adds an arbitrary number to RP is used to absorb the difference in the path length, so that instructions after control flow join can access the registers that were written before the branch, regardless of the path that is taken. For procedure calls, arguments and results are similarly set by the "register move" instruction. Because the logical register has enough space, most arguments can be passed using the register instead of stack access. STRAIGHT is provided with a rewritable SP so that any existing program can be transported to STRAIGHT by at least using SP and memory access.

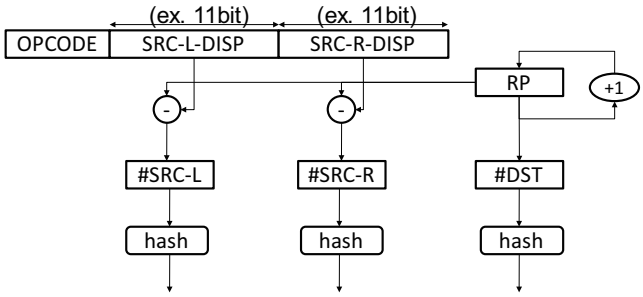

Fig. 6. Determining the Corresponding Nodes

B. Execution Model

Several instructions are fetched at once using branch prediction, which is similar to superscalar architectures. Subsequently, corresponding nodes of sources and destinations are revealed by hashing each register number (Figure 6). The width of the frontend pipeline is more easily increased because the rename stage is removed. The scheduler entry is assigned from the corresponding node of the source register L at the dispatch stage. An inquiry for a ready signal is sent to the scheduler of the corresponding node of the source register R. Each scheduler maintains and updates the ready flag of source L and R from the destination register number of issued instructions, and informs the issue candidates of this cycle to the select logic. The select logic is centralized, enabling efficient use of the compact datapath.

Each node is provided with an associative table for the distributed register file and its manager unit. Each register value is maintained using the register number as the key. While each table has number of entries, their access frequency is not high because most of the register access is absorbed in the bypass network. Fewer port numbers are therefore sufficient for the table.

The request for register allocation is sent to the corresponding node at the dispatch stage. As in the case of the cache algorithm, the node searches for invalid entry from the set and allocates the incoming register number. The distance between the stored register number and incoming register number determines whether or not the entry is invalid. Unlike the cache, the valid line cannot be evicted. When such conflicts occur, the next node handles the incoming register with a certain overhead. The manager unit periodically checks the tag array and searches invalid lines. Consequently, register allocation and free are performed as a localized operation at each node. The corresponding node of a register can be attained by hashing its register number so that broadcast and centralized mapping management are not required. For the hash function, simple XOR logic is sufficient as well as SHA-1.

STRAIGHT may have a longer register stage because it involves inter-node requests to the distributed register file and

the access of the associative table. However, the execution throughput is mainly determined by critical loops such as scheduler and result bypass [3], [27]. The size of the critical loop of STRAIGHT is sufficiently adequate to realize a high throughput that is comparable to conventional cores.

In STRAIGHT, the recovery from branch misprediction is relatively simple. First, the corresponding entries of the scheduler and long execution pipeline such as floating point arithmetics or load-store-unit are invalidated. Entries that correspond to a destination register number that is larger than the branch instruction are invalidated. Then, RP is rewound to the next destination register number of the branch instruction. The corresponding register values are not invalidated; however, they are eventually overwritten by the value of the correct path. For context switching, all entries of the distributed register file must be saved, in addition to PC, SP, and RP. Although the number is large, the transfer can be done gradually in background.

C. Further Optimization

The scope of this paper is to develop the first outline of the STRAIGHT architecture. However, its implementation of following optimization enhances the architecture feature. The associative table in each node can also be used to maintain the scheduler entry or cache line. By saving the scheduler entry in which the parent instruction does not appear to be quickly issued to the table, instruction window can proceed to the succeeding instructions while keeping the scheduler small. Also, register cache techniques are effective to mitigate the distributed register latency.

D. Features Enabled by STRAIGHT

To summarize the outline model, the following are the features that are enabled by the STRAIGHT architecture. First, note that each stage has the same or less complexity compared with conventional superscalar processors, enabling the clock frequency to be maintained. Moreover, because the rename stage is eliminated, the frontend latency is reduced. In addition, the frontend width can be easily extended.

STRAIGHT enables a large amount of logical registers. The compiler can exploit these registers to realize various optimization techniques. Spill access can be significantly reduced so that the inherent parallelism in the programs is easily exploited. STRAIGHT has wider instruction window to exploit the ILP, which is enabled by using decentralized register file and manager. Although the basic scheduler size is relatively small, the effective scheduler entry size is extended by exploiting the table of each node, as described above. The execution pipeline is sufficiently compact to ensure the speedy and efficient execution of the filled instructions.

STRAIGHT removes complex controlling and directly performs the execution with logical registers. This scheme achieves greater power efficiency because it can reduce the power required for control. For example, the hot spots of RMT and free list are removed. Instead, STRAIGHT requires larger tables for the register, however, they are not accessed as frequently as RMT. This scheme will be more effective when the devices have a feature whereby the active power significantly exceeds the static power, which is indicated by several device techniques [28].

V. Preliminary Evaluation

A. Performance Estimation

We conducted performance estimation of STRAIGHT by using a cycle accurate superscalar simulator. Onikiri2 rev.5321 is used as a baseline simulator, which faithfully implements the state-of-the-art superscalar pipeline structures [29]. The expected parameters of instruction window size, frontend latency, and register latency are used to approximate the performance. As the simulation is performed by superscalar model, this first evaluation does not include advantages and overheads which are derived by ISA features of STRAIGHT, still it approaxiates the effect of ILP exploitation by using huge instruction window which is enabled by our architecture.

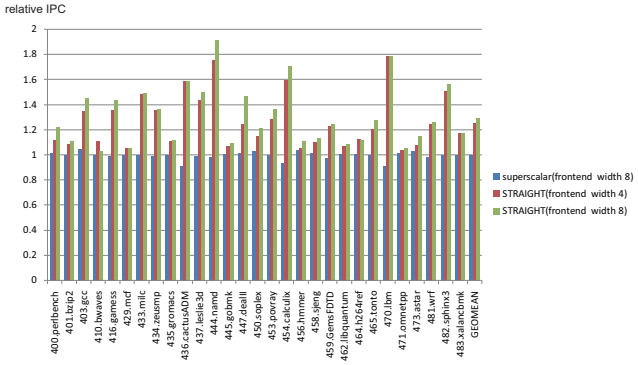

Fig. 7. Performance Improvement

The performance of real STRAIGHT will be increased by exploiting large amount of logical registers, which enables to eliminate spill registers and also allocate large data structures to the registers. On the other hand, the performance will be decreased by the code restriction of STRAIGHT, such as the overhead of write-onece policies, extra meory accesses which are caused by the shortage of the concurrent registers, and the overhead of register refreshing to avoid the register lifetime.

Table I shows the parameters of the baseline and STRAIGHT core. A 2k logical register and 512entries for each node (the node count is 8, thus in total 4k register entry; however, we conservatively estimated that only 2k are available because of the overhead of implicit free management) are assumed, because this is the first estimation, these parameters are yet to be optimized. Note that the sizes of the backend of the baseline and SRAIGHT are set to be the same. The memory subsystems are also the same, and we assumed the main memory latency to be a bit optimistic value of 50 cycles because of the incoming 3D memory stacking technology [30]. We evaluated all benchmark programs of SPEC CPU 2006. Each program was compiled using gcc version 4.2.2 with the -O3 option. A cycle accurate execution of 256 million instructions was simulated after skipping 10 billion instructions from the program head.

Figure 7 shows the scale merit of STRAIGHT, which estimates the effect of wide frontend and large instruction window. The x-axis shows the benchmarks and the y-axis shows the relative IPC compared with the baseline superscalar. The models of i) the superscalar with a wide frontend, ii)STRAIGHT (with the same frontend width as the baseline), and iii) STRAIGHT (with the frontend width of 8) are plotted. The results show that STRAIGHT can exploit more ILP, and has a 30% better IPC, which is in the geometric mean of all the SPEC2006 benchmark suite programs with the same backend size. The comparison also reveals that the improvement is achieved mainly by extending the register file and instruction window; however, the large frontend width is also effective for several benchmarks such as 447.dealII and 473.astar. In contrast, by increasing the frontend width the performance in a conventional superscalar is worsened.

Figures 8, 9, and 10 show the usage of functional units with the same settings as in Figure 7. The usage value shows “1” when all of the corresponding functional units were busy every cycle. As shown by the graph, the usage of functional units in superscalar processors is generally low. However, STRAIGHT

TABLE I
MICRO-ARCHITECTURAL PARAMETERS

	Baseline Superscalar	STRAIGHT
Instruction Set Architecture	Alpha AXP	
Front-end	4 way, 7 cycle	8 way, 5 cycle
Instruction Window	int 64, fp 32	int 512, fp256
Register	int 128, fp 128	int 2k, fp 2k
Functional Units	2 iALU, 1 iMUL/DIV, 2 LD/ST, 1 fpADD, 1 fpMUL/DIV/SQRT	
L1 I-Cache	32 KB, LRU, 8 way, 64 B line, 1 cycle latency	
L1 D-Cache	32 KB, LRU, 8 way, 64 B line, 1 cycle latency	
L2 I/D-Cache	4MB, LRU, 8 way, 64 B line, 12 cycle latency, with stream prefetcher	
memory access	50 cycle	

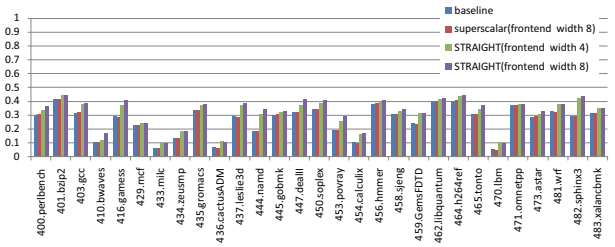

Fig. 8. Comparison of Functional Unit Usage (iALU)

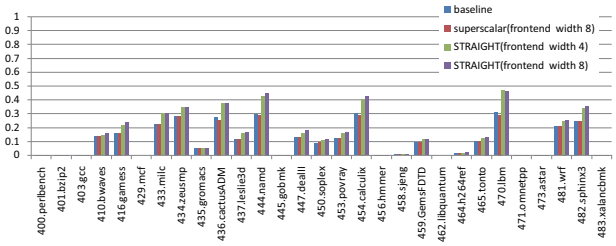

Fig. 9. Comparison of Functional Unit Usage (fp adder)

shows significantly higher usage, and thus effectively operates the execution units. In particular, the usage of the floating point units is significantly improved. These result reveal that the pipelines in a conventional architecture are not fully exploited; therefore, large instruction window of STRAIGHT improves the perfoemance without extending the piplines.

Next, Figure 11 shows the effect of the register latency in STRAIGHT. The y-axis indicates the relative IPC to baseline superscalar processor. The bars show the register latency from 4 to 8 cycles. In geometric mean, the performance is degraded about 2% for an additional cycle, still shows significant performance improvement even with 8 cycle latency. However, performance of several benchmarks is below the baseline at higher register latency. Mechanisms to mitigate latency are

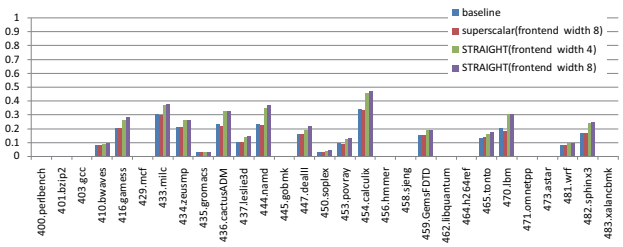

Fig. 10. Comparison of Functional Unit Usage (fp multiplier)

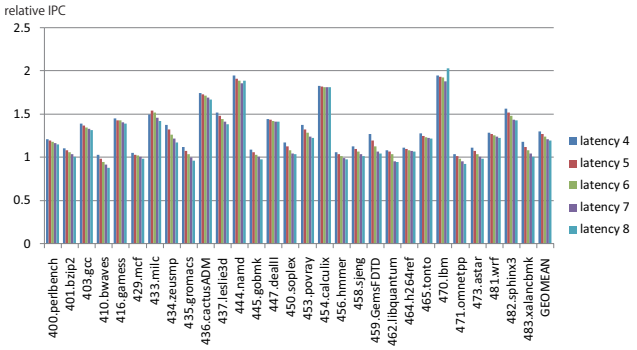

Fig. 11. Effect of Register Latency in STRAIGHT

required for these programs. The performance of 470.lbm is increased at latency 8 because of side-effect of latency predictors and prefetcher.

B. Hardware Budgets

Figure 12 shows the estimation of hardware budgets for the total capacity of the memory resources. The hardware parameters of Table I are used. Assuming that a 4MB L2 is shared by 4 cores, 1 MB is declared as the capacity of L2 in each core. For RMT, we assumed 8 check points. Figure 12 shows that for STRAIGHT the total capacity increases with the size of the register file, instead of RMT. For the bit count, the difference is as large as the cache size of D1. However, the port count of RMT is several times larger than that of the distributed register file of STRAIGHT. The port count increases the resource cost by a factor of 2, so the actual difference of the area or transistor count is reduced. When the capacity of L2 is included, the difference is negligible.

VI. CONCLUSION

This paper reveals a novel STRAIGHT architecture that efficiently improves the core performance for manycore processors. Inspired from the distributed key-value store scheme, STRAIGHT realizes a scalable distributed register file, which enables straight execution of the large logical register code. Each register is written once and is eventually invalidated, significantly reducing the cost of register management. As opposed to conventional clustered microarchitectures, STRAIGHT is not provided with a scalable execution pipeline; however, it has an efficient scale of the execution pipeline. By revisiting previous architectures, we observed that the usage of functional units is low, and distribution is therefore needed for the instruction window, instead of execution pipelines.

Being the first report on our new approach, this paper describes an outline of the software and hardware model. We also conducted a preliminary performance evaluation and cost estimation. It is expected that STRAIGHT will improve the ILP performance by 30% relative to that of a conventional

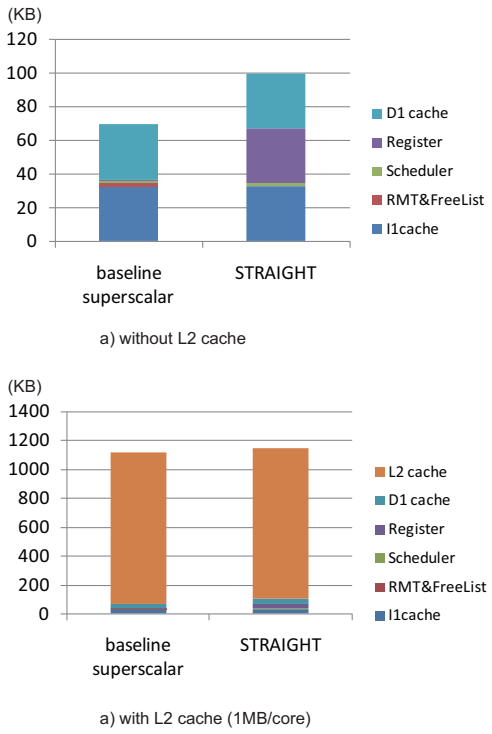

Fig. 12. Budgets Estimation

superscalar processor, while it does not require a significant additional cost overhead. Moreover, a reduction in register management can lead to efficient power reduction. We are now in the process of developing STRAIGHT compilers and an RTL specification for the core to verify in more detail performance parameters.

REFERENCES

[1] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," *Int. Symp. on Computer Architecture*, 2012.

[2] A. Nicolau and J. A. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. Comput.*, vol. 33, no. 11, pp. 968 – 976, 1984.

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors," *24th Int. Symp. on Computer Architecture*, pp. 1–13, 1997.

[4] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," *Int. Symp. on Computer Architecture*, pp. 414 – 425, 1995.

[5] G. A. .Kemp and M. Franklin, "PEWs: A decentralized dynamic scheduler for ILP processing," *Int.Conf.on Parallel Processing*, pp. 239–246, 1996.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Processor Cycle Time Through Partitioning," *30th Int. Symp. on Microarchitecture*, pp. 149–159, 1997.

[7] R. Canal, J. M. Parcerisa, and A. Gonzalez, "A Cost-Effective Clustered Architecture," *Int.Conf.on Parallel Architectures and Compilation Techniques*, pp. 160–168, 1999.

[8] J. E. Smith, "Instruction-Level Distributed Processing," *IEEE Computer*, vol. 34, no. 4, pp. 59–65, 2001.

[9] R. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[10] A. Baniasadi and A. Moshovos, "Instruction Distribution Heuristics for Quad-Cluster Dynamically-Scheduled, Superscalar Processors," *33rd Int. Symp. on Microarchitecture*, pp. 337–347, 2000.

[11] R. Canal, J. M. Parcerisa, and A. Gonzalez, "Dynamic Cluster Assignment Mechanisms," *6th Int. Symp. on High-Performance Computer Architecture*, pp. 132–140, 2000.

[12] J. M. Parcerisa and A. Gonzalez, "Reducing Wire Delay Penalty through Value Prediction," *33rd Int. Symp. on Microarchitecture*, pp. 317–326, 2000.

[13] B. Fields, S. Rubin, and R. Bodik, "Forcusing Processor Policies via Critical-Path Prediction," *28th Int. Symp. on Computer Architecture*, pp. 74–85, 2001.

[14] N. Hattori, M. Takada, J. Okabe, H. Irie, S. Sakai, and H. Tanaka, "Instruction Steering Algorithms Based on Issue Delay," *IPSJ Trans. on Advanced Computing Systems*, vol. 45, no. 11, pp. 80 – 93, 2004.

[15] H. Irie, N. Hattori, M. Takada, S. Sakai, and H. Tanaka, "Distributed Speculative Memory Forwarding for Clustered Superscalar Processors," *IPSJ Trans. on Advanced Computing Systems*, vol. 45, no. 11, pp. 94 – 104, 2004.

[16] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," *Int. Symp. on Microarchitecture*, pp. 40 – 51, 2001.

[17] M. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim, "Evaluation of the raw microprocessor: an exposed-wire-delay architecture for ilp and streams," *Int. Symp. on Computer Architecture*, pp. 2 – 13, 2004.

[18] C. Kim, D. Burger, and S. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *IEEE Micro*, vol. 23, no. 6, pp. 99 – 107, 2003.

[19] D. M. Tullsen, S. J. Eggers, J. S. Emer, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *22nd Int. Symp. on Computer Architecture*, pp. 392–403, 1995.

[20] J. D. Collins and D. M. Tullsen, "Clustered Multithreaded Architectures - Pursuing Both IPC and Cycle Time," *18th Int.Parallel and Distributed Processing Symp.*, pp. 76–85, 2004.

[21] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *Trans. on Computers*, vol. 48, no. 9, pp. 866 – 880, 1999.

[22] A. Roth and G. Sohi, "Speculative data-driven multithreading," *Int. Symp. on High-Performance Computer Architecture*, pp. 37 – 48, 2001.

[23] Z. Purser, K. Sundaramoorhy, and E. Rotenberg, "A Study of Slipstream Processors," *33rd Int. Symp. on Microarchitecture*, pp. 269 – 280, 2000.

[24] M. Date, N. Kurata, R. Shioya, M. Goshima, and S. Sakai, "Processor Architecture that Minimizes Register Renaming and Dispatch Network," *Symp. on Advanced Computing Systems and Infrastructures*, pp. 280 – 288, 2012.

[25] K. Horio, H. Hirai, M. Goshima, and S. Sakai, "Twintail Architecture," *Symp. on Advanced Computing Systems and Infrastructures*, pp. 303 – 311, 2007.

[26] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," *ACM SIGCOMM*, vol. 31, no. 4, pp. 149–160, 2001.

[27] R. Shioya, K. Horio, M. Goshima, and S. Sakai, "Register Cache System not for Latency Reduction Purpose," *Int. Symp. on Microarchitecture*, pp. 301 – 302, 2010.

[28] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, "Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 305 – 327, 2003.

[29] R. Shioya, M. Goshima, and S. Sakai, "The Design and Implementation of Processor Simulator "Onikiri2"," *the Annual Symposium on Advanced Computing Systems and Infrastructures, poster*, 2009.

[30] C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, "Bridging the processor-memory performance gap with 3D IC technology," *Design Test of Computers, IEEE*, vol. 22, no. 6, pp. 556 – 564, 2005.