Programming Scalable Service in Code segment and Data segment

Shoji TAMAKI
Information Engineering
University of the Ryukyus
Nishihara-cyo 1, Okinawa, 903-01, Japan
Email: shoshi@cr.ie.u-ryukyu.ac.jp

Shinji KONO
Information Engineering
University of the Ryukyus
Nishihara-cyo 1, Okinawa, 903-01, Japan
Email: kono@ie.u-ryukyu.ac.jp

***Abstract*—**
To implement scalable services, not only higher software design, low level implementation is also important to achieve performance and reliability. A combination of fine grain task manager and continuation based language is good to make Scalable Services on Many core architecture. Code segment is a small part of execution code written in a lower language of C. Data segments are fragments of memory and these are passed among code segments and CPU cores. We discuss the pro and cons of our method.

***Keywords*-scalable; continuation; parallel processing;**

I. Tools for implementing Distributed Application

On demanding construction of scalable services such as Twitter, FaceBook or Network based Book Publishing, we need new stage of programming tools. Based on our experiences, we designed and implemented two major tools to build scalable services: Code segments and Data Segments.

Not necessary mentioned SEDA [1], scalable services requires highly distributed servers and highly multi-threaded program on a server among them. This type of implementation works fine in theory, but it heavyly depends on low level implementation, such as threads, synchronized queues and CAS operations.

We have successfully implemented WWW services using Classical tools such as C++, Java, or even C. Script Languages such as Perl, PHP or Python are used in front end, but in case of heavy duty database side, careful implementation is necessary to achieve good performance.

Now some of the services have more than 10 millions users, load balancing among several WWW front-end and many memcached[2] servers to replicate Database accesses using classical database such as Oracle, mySQL or Postgress, which performs so badly, Internet companies have to create Key Value Store system by themselves, such as BigTable [3] or Cassandra[4]. This situation is sometimes discussed in a context of ACID vs BASE database scheme [5].

Based on our works on Internet programming and Sony PS3 programming, that is Cell architecture[6], now, we are sure that we need more suitable tools to implement various components in the scalable services, such as database server, web server or HTTP response generator even in a front end.

We separate difficulties in two point of views: Code and Data. This sounds very basic, but since our history is starting from a single CPU with few memories, current tools are some how obsolete now, so we have to reconsider the situation (Fig. I).

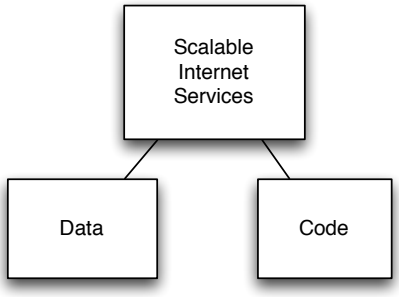

Figure 1. Data and Code in Internet Service

We are working on a combination of Continuation based C[7], [8]

and

Cerium Engine[9]. The former one is a lower language of C implemented in GCC[10], and the later is a Open CL[11] like fine grain task manager on PS3 Cell architecture, which supports data segment management on SPE (Synergistic Processor Engine). Since SPE has only 256Kbytes local memory, careful management is necessary, so we have to invent our own memory manager. We can use 6 SPE with 2Tbit/s ring bus in PS3 Linux (Fig. I).

In this paper, first we analyze problems in scalable system. Then we introduce new concepts: Code Segment and Data Segment. Code segment is implemented in Continuation based C here after we call CbC and Data Segment is implemented in a memory management module in many core task manager called Cerium Engine.

The basic idea is this. We pass the control among module layer without function call. We cannot use conventional language because it has built-in function call which which

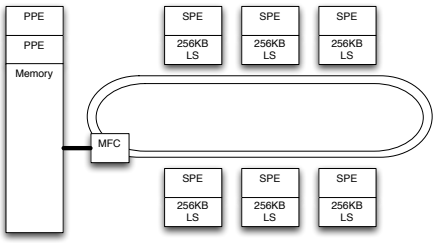

Figure 2. Cell Architecture

cannot be removed. These module layer segments are called code segments. All the data are stored in a message packed from, which we called data segment, which is controlled uniformly. Instead of using direct pointer access, data segments are copied among modules and CPU cores, which are carefully adopted to the cache or interconnect communication such as DMA. All the data segments are hashed in 2^n size memory pool similar to the Unix malloc mechanism. This pool is in 64bit address space and it makes data segment communication far simpler.

It sounds like completely different from current Internet service scheme, we overcome the difference in following way using code transformation. First we make entire program in a conventional way. We divide it into code segment and explicit stack manipulation. During this stage, communications are reformatted into data segment passing among code segments. The program still working exactly the same before transformation and we may use automatic conversion here. We reorganize it using data parallel and pipeline execution. At this stage, automatic conversion is not suitable in many cases, so we have to make translation by hands, but we can use possible equivalence checker for the program correctness. In order to make pipeline execution, destructive modification of the content of data segments such as classical object oriented programming is not allowed. We have to make copies.

At the last section, we give some of our achievement and comparison with other tools, such as SEDA or Open CL.

II. Problems

Let's think that we are going to make a network game. Maybe PS3 (6 SPU and 2 PPU)is used in a client side and 32 CPU (16 x 2 hyper thread). We have to use highly pipelined thread and data parallel execution in both client side and server side, something like SEDA architecture. We will demonstrate several problems based on our experiences.

A. Our Experiences

Our PS3 implementation is SPURS[12] like Pipeline Task Manager which is called Cerium. (We had to write software rendering because of SCEI did not open GPU information BTW.) Since PS3's SPU has only 256Kbytes memory, we have to carefully handle memory usage both in case of code and data.

Data segment is copied from PPU to SPU via DMA, which overhead is hide using Task Pipeline. But we have to avoid too much synchronization of these copies.

In case of Xeon architecture, CPU cores are shared all the memory, but actually it has a local cache which is interconnected using quick path. Cache size is 256Kbyte for each core. Implicit copy is done between a cache memory and the main memory or between a cache and another cache. The situation is basically the same in PS3 and Xeon.

B. Module Layer

Complex systems such as Operating systems, Database systems, Network Systems or Game libraries have several module layer. For examples, Database system has message packing module, query analyzing module and execution module. Network system may have ISO standard layer such as presentation, transport and network. Operating system have v-node file system and device drivers.

Each module may have 1-5 nested function calls, so we have more then 10-30 nested function calls in Complex systems. It can be implemented in normal function calls (Fig.II-B).

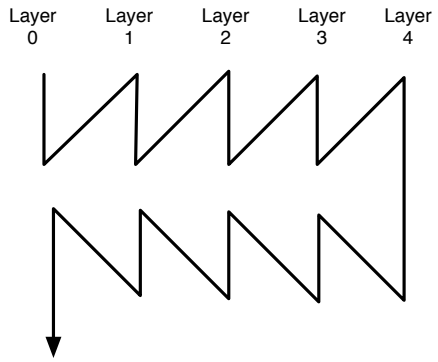

Figure 3. Layer by call

Using our Continuation base C, layering can be implemented in a goto statement. Since it has no stack operation, it works very fast (Fig.II-B).

If we have several tasks to do, each processing in modules can be executed in a pipelined way. In order to implement the pipeline, we assign threads from thread pools to each module layer . Each thread is interconnected by a synchronized queue, which has certain overheads, but if it is carefully implemented, parallel processing hides its costs.

Conversion from sequential execution to pipelined execution is straight forward, but if it has a race condition, correcting the problem is very difficult.

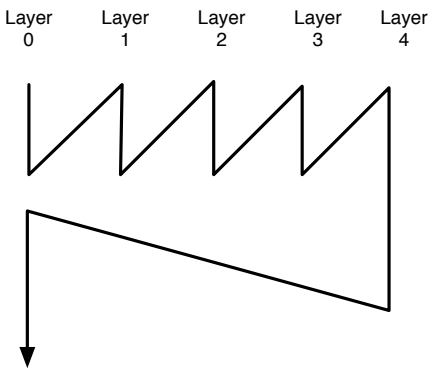

Figure 4. Layer by goto

C. SEDA

This combination of pipeline staging and data parallel execution is the heart of SEDA. But it requires very complex programming. At first we have design communication queue among all the pipeline stages. In case of C++, we have to managing all the queue manually because it lacks garbage collection. It is not so easy and requires complex memory pools (or conservative GC), which is a bug prone (Fig.II-C).

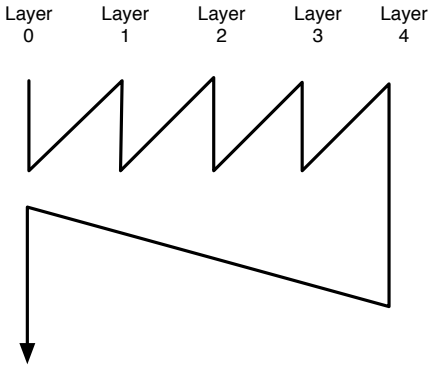

Figure 5. Layer by Thread

D. Thread Implementation

Theoretically SEDA architecture works fine, but it assumes very fast thread execution with blocking queue. Cassandra key value store system use Java to implement this architecture. Java 6 is far better thread execution, but sine it is a combination of user level thread and kernel level thread, it is not so easy to optimize its execution.

If we use script language to implement this, thing become worse. For example, Python thread implementation is very bad concurrency[13], and ruby does not support kernel thread. And their GC mechanism always interfere with thread executions.

E. Blocking queue

Each threads are executed in an event driven way. A task is put into a blocking queue and it wakes up a thread. The thread read the queue atomically.

We can write this operation in following way.

```
    while(Task p = waitingQueue.get()) {
p.run();
    }
```

It looks good but `p` is determines just before its execution, which is no good in terms of branch prediction. It looks like this delay is not so important, but it has penalty around 10 clocks. If we have many small task this situation is not so good. What we need is 10 to 20 instruction cycle before executing the indirect call.

Besides blocking queue's CAS costs, queue operations include allocation / deallocation cost. In case of Java, to avoid GC penalty, link node is not reused and it simply delete old one and create new one. If the new operation is shared among threads (unlikely), it requires another CAS, otherwise it requires separate memory pool for each thread, which consumes a lot of memory.

F. Scheduling

If cost of blocking queue operation is negligible, simple FIFO scheduling is OK in SEDA from the through put point of view. But blocking queue requires 10 to 20 instruction cycle under no race condition. Thread pool size is heavyly depends on the architecture, that is number of CPU, number of requests, execution time of tasks.

Sometimes it is better to reduce concurrency and skip these synchronization costs. In this case, synchronization of threads becomes just a cost.

G. Garbage Collection

Basically communication between layers makes no garbage, because it is generated and destroyed in fix amount size. But in case of programming language with GC, if we use memory pool like technique, it makes many references which GC have to take care of. It makes GC very slow. So it is better to simply generate and destroy.

Apache Web server features memory pool approach drastically, but it is an convention, some module use malloc library call directly.

H. Programming Correctness

SEDA architecture or SPURS architecture is very complex to implement and the program working on it is very difficult also. It is very difficult to test.

For example, message packet between pipeline stages is created and destroyed in exact moment. If we lost the correct

timing, a bug will occur or not if we are unlucky. This is odd, because even if program itself is deterministic, it behaves non deterministic dew to pipeline execution timing.

I. Many Core Awareness

Open CL [11] is a standard library to use Many Core architecture, but it has very complex interfaces. We have to write a program on a core in a string, which is compiled in LLVM [14]. Data transfer API is vary and complex, which requires large amount of code.

In case of Java or Scripting language, we cannot directly control the copy between cores, which means we cannot hide copy cost explicitly. We have to care about SPU's local memory size or cache memory size which is 256Kbytes in this time.

The same careful management is necessary for executing code which is a data on a core also. We have to transfer code segment using copying cost hiding technique such as pipeline execution.

These higher level pipeline optimization is very difficult and is not handle well in compiles. Since compiler technique is working well on streaming instructions, it is some how contradict. It should be designed by hands.

III. New Tools

We introduce two main tools, one is Continuation based C and the other one is Data segment management based on Cerium Task Manager.

IV. Code Segment

Continuation based C is a C language which all the function is forced to do tail call elimination. It is implemented using GCC 4.x. Modification is not so large. We also force FASTCALL option which assign arguments on registers. This makes it faster.

CbC Syntax is very simple.

```
struct interface1 { DataSegment<Data> *i;};
struct interface2 { DataSegment<Data> *o;};

__code f(struct interface1 *a,
struct interface2 *b) {
      b->o=a->i;
      goto g(b);
}
```

In this example, a code segment `f` has `input a` and sends `output b` to a code segment `g`. There is no return from code segment `b`, `b` should call another continuation using `goto`. Any control structure in C is allowed in CbC language, but we may restrict ourselves to use `if` statement only, because it is sufficient to implement C to CbC translation. In this case, code segment has one input interface and several output interfaces (fig.IV).

`__code` and parameterized global goto statement is an extension of Continuation based C. Unlike `C--` [15]'s parameterized goto, we cannot goto into normal C function because of forced FASTCALL option.

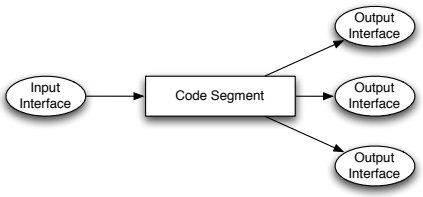

Figure 6. Code Segment

A. Continuation

Since code segment has no stack, continuation of code segment is mere entry address to the code segment. We can call it a light weight continuation.

We also supports full continuation of normal C function using GCC nested function and statement expression. It is implemented some like this in GCC compiler in a pseudo code with GCC extensions.

```
void (*__return)(int retval_, void *_envp);
  __return = ({
     nee_label__ _cbc_exit0;
     void __return_func(int retval_,
                  void *_envp){
          retval = retval_;
         goto exit0;
     }
      if (0) {
        _cbc_exit0:
         return retval;
     }
      __return_func; // return value
    });

    void *__environment =
       __builtin_frame_pointer();
```

We have a environment pointer which is usually the frame pointer, but it is not used here, because this is a closure with a hidden environment. Since this closure is usually implemented using trampoline, that is executable code on stack, if execution code on stack is prohibited, it will not work, but it works on Linux and Mac OS X. In case of Windows case, we cannot use closure, so we have to assign frame pointer explicitly. If we don't have to handle frame pointer directly, generation of continuation is done in parsing phase. This is important to make GCC modification minimum.

Anyway this can be used like this.

```
int main() {
  goto f(1, __environment, __return );
    ....
}
```

```
__code f(int, void *env,
  __code (*continuation)(int retval_,void *fp)) {
goto (continuation)(-1, env);
}
```

In this example, `main` will return -1. When you want to return to the middle of the normal function or code segment, put an extra function call over it.

V. Data Segment

We have Open CL like task manager with data segment.

Data segment is a set of doubly linked fix size block which also hashed by the 64bit address. It has 2^n size, so it is allocated efficiently.

Each site, CPU or cores expected to have separated data segment pool. Data segment address is unique in all CPUs. In case of PS3, SPU has local storage that is 256Kbytes separate addressing space, which local address is different from data segment global address.

Code segment will not use global address directly but it will use offset in data segments in its input interface. So we can use same code segment both for 64 bit Xeon and SPU 256Kbytes memory. This means data segment size is normally limited by its hardware, Typically 16Kbytes (fig.V).

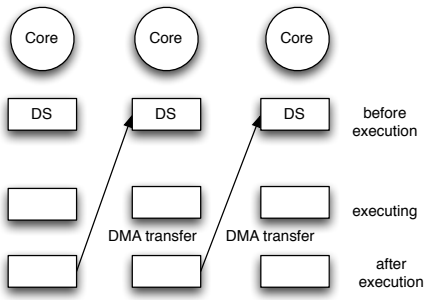

Figure 7. Pipeline buffered data segment

Each Core have to have two input segments and two output segments to make pipeline correctly. With two extra segments are necessary for task array it-selves, so we have 6 segments total.

A. Data Segment operations

Data segment has several operations,

```
get with no global address
get with copy
get with no copy
get with copy with write back
get with no copy with write
```

Allocation/deallocation is not directly handled from its code segment, because it is handled by the Task Manager in a pipelined way.

API can be called from a Task like this,

```
{
  Datasegment tile =
    smanager->get_segment(addr);
```

but usually it not visible from the task, because its reading operations were done before its execution and its writing execution will be done after the task execution.

Data segment may contains other data segments' global address, but it may invalid. It is a kind of key in a key value store. Consistency of data segment global address is maintained by the Cerium task manager.

VI. Typical usage

A code segment is provided input interface, which contains array of data segment in local address space or cache. Usually availability of data segment is assured by the task manager. If it is not ready, the code segment waits and other ready-to-run code segments are executed.

Loading necessary data segments in the input interfaces are done prior to the execution, may be in a back ground of other code segments execution.

In following example, `t_exec` is created, and it has one input data segment and one output data segment. It can be executed in any SPU (PS3's CPU core), and `t_print` task have to wait for its completion. Finally it is spawned.

```
HTask *t_exec =
   manager->create_task(TASK_EXEC);
t_exec->add_input_datasegment(i_data);
t_exec->add_output_datasegment(o_data);
t_exec->set_cpu(SPE_ANY);
t_print->wait_for(t_exec);
t_exec->spawn();
```

When all data segments are ready, the cod segment is executed. During its execution, next input interfaces may be loading.

After the execution, output interfaces are written into the global address if necessary. This is also done in a pipelined way.

A. Task dependency and Task array

Cerium task manager has very simple FIFO scheduler. It is sufficient if only through put is matter, which is a usual case.

All the task is stored in data segments, and connected wait-for link.

After a task is finished, the task manager solve these dependencies, which is a rather heavy task. If tasks are grouped in terms of dependencies, we can reduce this phase. This is called task array.

All the data is stored in data segments and it is managed in data segment pool in each separate CPU, that is we need no lock in its allocation.

B. Task execution

After the loading of input interface, if we have a next task, we know where to execute it. It can be passed to the current task.

```
    _code task_a(next_task, interface input,
           interface output) {
.... Task processing
        goto next_task->code(next_task,
          next_task->input, next_task->output);
    }
```

If we have not task to execute more, we can put mail waiting task in the `next_task`. In this way, `next_task` call address is determined well before the call.

C. Data segment deallocation timing

There are two types of data segment.

The one is staying in a main memory indefinitely, possibly replicated in more reliable storage hierarchy such as SSD or Hard Disks. It's global address is persistent. It is basically write only and remain forever in the life of the Internet service. In other words, it will never be deallocated. We can call this a persistent data segment.

The other data segment is stayed in local cores. This is limited and temporally. It is copied from the persistent data segment. After the code segment execution, temporal data segment may be copied into persistent data segment.

Task itself don't care about reading and writing race conditions. It have be controlled in terms of task dependency or be controlled by the task manager (fig.VI-C).

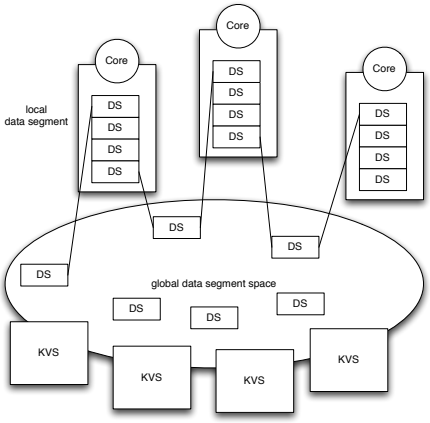

Figure 8. Global and local data segments

D. Where is the synchronization?

In Cerium Task Manager, a core has a single threaded scheduler. It accept an array of task as a data segment as a mail. There is a main task manager, which waits mails from schedulers in cores. Synchronization only happens in mail transfer among main scheduler and sub schedulers in cores. This means synchronization itself can be delayed significantly in this scheme.

If some service needs very fast response, dominating special task is necessary. For example, it wait some events using spin locks or hardware interrupts.

E. Hardware support

This architecture requires explicit cache control. But now a day, most architecture has this kind of cache control such as memory barrier. Unfortunately these are not standardized. Using Cerium task manager, we can hide these differences.

F. With Conventional Operating System

Task Manager itself is running in a user space. Since tasks are in data segments and it can be transferred to other user spaces, for example in other clusters. Actually we build our task manager in user space.

There are possible operating system supports for this task manager or we can provide memory space management for code segments and data segments.

G. Object Orientation

There are many object oriented programming style since Smalltalk-80[16]. C++ or Java has an object in fixed memory address. When a field of an object is updated, fixed memory contents is updated. In case of highly pipelined execution, updating memory contents requires synchronization when the object is shared.

In our scheme, usually input interface and output interface point different data segment to avoid synchronization.

In ACT3[17], actor has a become operator. An object is replaced by newly created object. This means object has multiple memory address according to its update history. In Smalltalk-80, it has object table and become operator is a replacement of pointer in the list. The list should be kept in a data segment and update by a single threaded task.

We can build actor like object oriented system on top of data segment pools.

H. Verification

Basically pipelined tasks are in fact, series of application of tasks on requests. We can simply writes this using iterator. In case of word count in a file,

```
foreach data segment d
     in ( file ),
     out in (partial_result) {
         task_work_count(d,out);
}
task_sum_up(partial_result);
```

If pipeline execution is correct, we don't have to verify the pipeline execution, but check the correctness of this sequential execution, which is much easier.

Once we get verified sequential execution, we can put checking stage on each pipeline stage.

VII. Comparison

Our architecture is a variant of SEDA, but it can reduce synchronization cost. Using data segment copying, shared data are reduced. Copying costs are hide using DMA or cache management instruction.

Open CL is recently introduced but it has very complex data transfer operation. It is an assembler level description to achieve best performance. Data segment handling makes it simple both in syntactically and in memory management efficiency.

Tasks in Open CL is stored in C strings. In our scheme tasks are all written in code segments, which can run on any architecture. Actually we can run Cerium both on Mac OS X and PS3 Linux using the same code. If it contains an optimized code for SPU, we can run a code has the same behavior with non optimized code.

Script language is easy to describe, but it works sequentially from the beginning. Python cannot achieve parallel processing advantage because of interpreter restriction. Our approach is that divide problem into code segments and data segments and execute it in an iterator.

This is something like FP[18], but using data segments, its execution becomes suitable for Many core architecture.

KVS, key value store is a distributed database which is separated from main service program, In our scheme, management of global data segment can be done in a KVS. We can also use our architecture to implement a KVS.

In Persistent programming, records and transactions are introduced in Programming Languages, but at that time, parallel execution is not well considered. In our scheme, data segments behave as records, which has several versions.

This architecture uses a new language CbC. It is a lower level language of C, but still programmers have to learn it. It is very different programming style also. We think it is not so easyly accepted by every one.

In structured programming and data flow diagrams [19], everything has record like structure, which is called container. In this architecture, containers has common operations and managed in many core architecture. It is also executed in CbC.

There is a conversion algorithm from C to CbC, so we hope some kind of half automatic conversion of sequential implementation of the Internet service is possible.

VIII. Conclusions

We are developing SEDA like architecture for software service architecture. It has code segment system based on Continuation based C and data segment system based on Cerium Engine.

Combination of data segment and code segment provides a better many core programming than Open CL. It is executed in multi stage pipeline. Code segment provides good implementation technique of pipeline scheduler.

Data segment copying makes garbage collection unnecessary in computation pipeline which cannot avoid in case of GC based language such as Java.

Cerium task engine and Continuation based C compiler is developed openly and working. The combination of code segment and data segment is under construction.

References

[1] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 230–243, 2001.

[2] L. Phillips and B. Fitzpatrick, "Inside livejournal's backend," in *LISA 04*, 2004.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *IN PROCEEDINGS OF THE 7TH CONFERENCE ON USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - VOLUME 7*, 2006, pp. 205–218.

[4] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network," in *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2009, pp. 5–5.

[5] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[6] S. Corporation, "Cell broadband engine architecture," 2005.

[7] Shinji Kono , "Implementing Continuation based language in GCC ," in *Continuation Festa 2008*, April 2008.

[8] Shinji KONO, "CbC," March 2008. [Online]. Available: https://sourceforge.jp/projects/cbc/

[9] ——, "Cerium," March 2008. [Online]. Available: https://sourceforge.jp/projects/cerium/

[10] Free Software Foundation, Inc., "GCC, the GNU Compiler Collection," March 2008. [Online]. Available: http://gcc.gnu.org/

[11] A. Munshi, *The OpenCL Specification Version: 1.0*. Khronos OpenCL Working Group, 2009.

[12] K. Inoue, "Spu centric execution model," 2006.

[13] D. Beazley, "Inside the python gil," in *Python Concurrency Workshop*, 2009. [Online]. Available: http://www.dabeaz.com

[14] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[15] N. Ramsey and S. P. Jones, "A single intermediate language that supports multiple implementations of exceptions," in *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[16] Xerox, "The smalltalk-80 system," *Byte*, vol. 6, no. 8, Aug. 1981.

[17] G. Agha and C. Hewitt, "Concurrent programming using actors," in *Object-Oriented Concurrent Programming*. MIT Press, 1987.

[18] J. Backus, "Can programming be liberated from the von neumann style? a functiional sytel and its algebra of programs," *C. ACM*, vol. 21, no. 8, pp. 613–640, 1978.

[19] P. D. Bruza and T. van der Weide, "The semantics of data flow diagrams," in *In Proceedings of the International Conference on Management of Data*. McGraw-Hill Publishing Company, 1993, pp. 66–78.