2009 Ninth International Conference on Intelligent Systems Design and Applications

An Efficient Solver for Scheduling Problems on a Class of Discrete Event Systems Using CELL/B.E. Processor

Hiroyuki Goto
Department of Management and Information Systems Science
Nagaoka University of Technology
Nagaoka, Niigata, 940-2188 Japan
e-mail: hgoto@kjs.nagaokaut.ac.jp

Atsushi Kawaminami
Faculty of Management and Information Systems Engineering
Nagaoka University of Technology
Nagaoka, Niigata, 940-2188 Japan
e-mail: s063314@ics.nagaokaut.ac.jp

***Abstract*—This research implements an efficient solver for scheduling problems in a class of repetitive discrete event systems using a CELL/B.E. (CELL Broadband Engine). The essence of this involves efficiently computing the transition matrix of a system whose precedence constraints regarding the execution sequence of jobs can be described by a weighted DAG (Directed Acyclic Graph). This means solving the longest path problem efficiently for all pairs of source and destination nodes. For the first step towards a high-speed computation, we utilize SIMD (Single Instruction Multiple Data) functions.**

***Keywords*-CELL/B.E.; repetitive discrete event systems; directed acyclic graph; SIMD; transition matrix;**

I. INTRODUCTION

In this research, we implement an efficient solver for scheduling problems in a certain class of repetitive discrete event systems using a CELL/B.E. [1], [2] processor. The focused systems are flow-line style where the same facilities are used repeatedly. We assume that the relationships between the execution sequences of jobs and the occupation times in facilities can be represented by a weighted DAG. Typical systems include: production systems [3], transportation systems [4], etc.

It is known that the behavior of this kind of system can be formulated by linear equations called state equation in max-plus algebra [5], [6], a class of discrete algebra. The state equations in this algebra provide the earliest and/or latest event occurrence times. They include a state vector that represents the state of the system, a transition matrix that reflects the propagations times of events, and an input vector that supplies the feeding times to the system. The bottleneck of the solver is in calculating the transition matrix.

For a system whose precedence constraints can be expressed by a DAG with n nodes and m arcs, the time complexity for computing the transition matrix based on a simple method is $O(n^4)$. On the other hand, more efficient methods for computing the transition matrix with a time complexity of either (a) $O(n\cdot(n+m))$ or (b) $O(n^3)$ have been proposed in [7]. Method (a) is based on an adjacency list, whereas method (b) uses an adjacency matrix. Since $m \le n\cdot(n-1)/2$ is followed for systems with a DAG structure, method (a) appears more efficient. However, in systems with dense adjacency matrices, the overhead for generating the adjacency lists is relatively high, in which case method (b) may be more efficient. In addition, method (b) is predicted to achieve a remarkable reduction in computation time when utilizing processors with vector instructions that allow multiple elements to be computed simultaneously.

An approach to these functions is using the set of SIMD instructions [8] with SSE extensions that has become common in recently released Intel-compatible processors such as Pentium 4 and Athlon 64, etc. However, the set of relevant instructions have been extended repeatedly, which now requires advanced techniques for implementation and is time-consuming for maintenance. By contrast, there are alternative processors, CELL/B.E., to make use of the benefits of SIMD instructions. In recent years, reasonable CELL/B.E. processors have been released and installed in several portable PCs and home-use game machines. Since CELL/B.E. processors are designed for vector calculations from the initial version, a unified and user-friendly interface can be utilized for using SIMD functions.

Therefore, this paper focuses on a CELL/B.E. processor and examines the effect of speedup in the computation of the transition matrix for systems with a DAG structure. A Sony Playstation III (TM) equipped with a CELL/B.E. processor is used in this study.

II. MATHEMATICAL BACKGROUND

Denoting the real field by $\boldsymbol{R}$, define $\mathcal{D} = \boldsymbol{R} \cup \{-\infty\}$. If $x, y \in \mathcal{D}$, the following basic operators are defined: $x \oplus y = \max(x, y)$, $x \otimes y = x + y$ and $x^{\otimes y} = x \cdot y$. Let the unit elements for operators $\oplus$ and $\otimes$ be $\varepsilon\,(=-\infty)$ and $e\,(=0)$, respectively. If $m \le n$,

$$\bigoplus_{k=m}^{n} x_k = \max(x_m, x_{m+1}, \cdots, x_n).$$

In the representation of matrices, $[\boldsymbol{X}]_{ij}$ stand for the (i, j) th element of matrix $\boldsymbol{X}$, and $\boldsymbol{X}^T$ represents the transposed matrix. If $\boldsymbol{X}, \boldsymbol{Y} \in \mathcal{D}^{m\times n}$, $\boldsymbol{Z} \in \mathcal{D}^{n\times l}$,

$$[\boldsymbol{X} \oplus \boldsymbol{Y}]_{ij} = \max([\boldsymbol{X}]_{ij}, [\boldsymbol{Y}]_{ij}),$$

978-0-7695-3872-3/09 $26.00 © 2009 IEEE
DOI 10.1109/ISDA.2009.16

$$[\boldsymbol{X}\otimes\boldsymbol{Z}]_{ij}=\bigoplus_{k=1}^{n}([\boldsymbol{X}]_{ik}+[\boldsymbol{Z}]_{kj}). \tag{1}$$

Let the unit matrices for operators $\oplus$ and $\otimes$ be $\boldsymbol{\varepsilon}$ and $\boldsymbol{e}$, respectively. $\boldsymbol{\varepsilon}$ is a matrix, the elements of which are all ε, and $\boldsymbol{e}$ is a matrix, of which the diagonal elements are e and all off-diagonal elements are ε. The priority of operator $\otimes$ is higher than $\oplus$, and it is omitted when no confusion is likely to arise.

Assuming that the number of facilities is n, we consider the behavior of the k th job. Let the processing completion time of job k be $\boldsymbol{x}(k)$, and assume that its minimum value is supplied by $\boldsymbol{u}(k)$. Furthermore, let the processing time in each facility and the list of preceding facilities of facility i $(1\le i\le n)$ be $\boldsymbol{d}(k)$ and $\mathcal{P}(i)$, respectively. Under these assumptions, the earliest processing completion times of the corresponding job $\boldsymbol{x}_E(k)$ can be calculated using the following equation:

$$\boldsymbol{x}_E(k)=\boldsymbol{A}_k\otimes[\boldsymbol{x}(k-1)\oplus\boldsymbol{u}(k)], \tag{2}$$

where

$$\boldsymbol{A}_k=(\boldsymbol{P}_k\boldsymbol{F}_0)^*\boldsymbol{P}_k,\ \boldsymbol{P}_k=\mathrm{diag}[\boldsymbol{d}(k)],$$
$$[\boldsymbol{F}_0]_{ij}=\begin{cases}e:\text{if } j\in\mathcal{P}(i),\\ \varepsilon:\text{if } j\notin\mathcal{P}(i).\end{cases}$$

Equation (2) is referred to as the state equation, $\boldsymbol{A}_k$ as the transition or system matrix, and $\boldsymbol{F}_0$ as the adjacency matrix. Operator * is referred to as the Kleene star operator. For systems with a DAG structure, if the adjacency or weighted adjacency matrix is given by $\boldsymbol{X}\in\mathcal{D}^{n\times n}$, $\boldsymbol{X}^*$ is calculated as:

$$\boldsymbol{X}^*=\bigoplus_{l=0}^{s-1}\boldsymbol{X}^l=\boldsymbol{e}\oplus\boldsymbol{X}\oplus\cdots\oplus\boldsymbol{X}^{s-1}, \tag{3}$$

where $\boldsymbol{X}^{s-1}\neq\boldsymbol{\varepsilon}$, $\boldsymbol{X}^s=\boldsymbol{\varepsilon}$ $(1\le s\le n)$. s is an instance that depends on the precedence constraints of the system. In terms of graph theory, calculating the Kleene star is equivalent to solving a kind of the longest paths problem.

For a given adjacency matrix $\boldsymbol{X}\in\mathcal{D}^{n\times n}$, efficient algorithms for calculating $\boldsymbol{X}^*$ are proposed in [7]. These include the following two or three steps.

(1) Topological sort

If node j is located upstream of node i, represent this as $j\prec i$. The topological sort [9] aligns the nodes to satisfy $\mathcal{S}(j)<\mathcal{S}(j)$ if $j\prec i$, where the index of node i is represented by $\mathcal{S}(i)$. As it is based on a DFS (Depth First Search) method, the time complexity is $\mathrm{O}(n+m)$. Note that the result is not unique and depends on both the initial parameters and implementation.

(2) Create an adjacency list (only for method (a))

If node i is a preceding and adjacent node of node j, denote this precedence constraint by $i\to j$. Then, for a given destination node j, obtain the set $\mathcal{P}(j)$ of source nodes i that satisfy $i\to j$. This is done by obtaining the set of i that satisfy $[\boldsymbol{X}]_{ij}\neq\varepsilon$ for a given j, and repeating the same procedure for all j $(1\le j\le n)$.

(3) Iterative calculation of $\boldsymbol{X}^*$

After preparing a working matrix $\boldsymbol{Z}\in\mathcal{D}^{n\times n}$ for computing $\boldsymbol{X}^*$, initialize this to $\boldsymbol{Z}=\boldsymbol{e}$. Then, we update the value of $[\boldsymbol{Z}]_{ji}$ topologically from upstream nodes to downstream nodes. In method (a), this update is performed by creating an adjacency list $\mathcal{P}(j)$ in the following manner:

$$[\boldsymbol{Z}]_{ji}\leftarrow[\boldsymbol{Z}]_{ji}\oplus\max_{l\in\mathcal{P}(j)}([\boldsymbol{X}]_{jl}+[\boldsymbol{Z}]_{li}). \tag{4}$$

On the other hand, method (b) updates the value using an adjacency matrix $\boldsymbol{X}$, as follows:

$$[\boldsymbol{Z}]_{ji}\leftarrow[\boldsymbol{Z}]_{ji}\oplus\bigoplus_{l=1}^{n}[\boldsymbol{X}]_{jl}\otimes[\boldsymbol{Z}]_{li}. \tag{5}$$

For an instance i, the time complexities of (4) and (5) are $\mathrm{O}(n+m)$ and $\mathrm{O}(n^2)$, respectively. By repeating this procedure for all i $(1\le i\le n)$, the time complexity of computing the transition matrix is $\mathrm{O}(n\cdot(n+m))$ in method (a), and $\mathrm{O}(n^3)$ in method (b).

As mentioned above, if $\boldsymbol{X}$ is the weighted adjacency matrix of a DAG, the theoretical time complexity of method (a) is lower than that of method (b). However, (4) requires creating adjacency lists to calculate the second term of the right hand-side, whereas the corresponding term in (5) only requires max and '+' operations for fixed size row and column vectors.

Various processors designed for fast computation can calculate fixed size arrays very quickly. Thus there may be several cases where the algorithm based on (5) is faster. This is confirmed in subsequent sections.

III. CELL BROADBAND ENGINE

We overview the structure of a CELL/B.E. installed in the Sony Playstation III. The CELL/B.E. consists of a PPE (PowerPC processor Element) and a SPE (Synergetic Processor Element). The former has an all-purpose processor core, while the latter includes a specialized core for calculation. Fig. 1 gives an outline of this structure.

The PPE is an all-purposed processor with a 64 bit PowerPC architecture. It controls the SPEs as well as running the operating system. A set of vector calculations called 'Altivec' is available. It has, however, frequently noted that the processor is not suitable for floating-point calculations.

The SPE is a processor with a 128 bit SIMD architecture which allows for simultaneous operations on multiple elem-

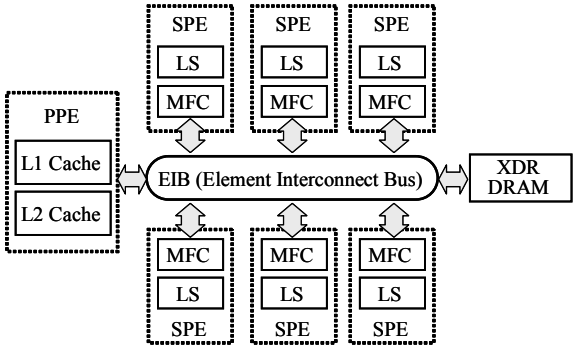

Figure 1. Internal structure of CELL/B.E.

ements. For computation of floating-point values of 32 bits, it can handle four elements simultaneously, while it can handle sixteen elements for 'char' type values of 8 bits. As such, it is expected that an elaborate optimization can achieve a remarkable reduction in computation time for vector and matrix calculations. On the other hand, the processor is not suited to the calculation of single elements, that is, of scalars.

For the main memory of the entire processor, fast accessible memory called XDR (eXtreme Data Rate) DRAM is used. In addition to this, each SPE has an internal RAM called the LS (Local Store) independently. This is a kind of L2 cache, with the result that each SPE can access its own LS very quickly. Data transfers between the LS and XDR are performed using the DMA interface called the MFC (Memory Flow Controller). In the case of a CELL/B.E. installed in the Playstation III, thre is only a single PPE with an operating frequency of 3.2 GHz. The number of SPEs available in Linux is six, and the size of each LS is 256KB.

IV. IMPLEMENTATION

In this research, we use 'float' variables of 32bits for storing data for $\mathcal{D}$. A special value '-FLT_MAX' is used to represent ε. As mentioned in the previous section, SPEs perform basic operations using 128 bit registers. For 'float' variables, four values can be computed simultaneously. However, the same registers are used even for operating on scalars. This means that, for an operation on five elements, for example, requires two SIMD operations. Moreover, it should be noted here that calculating the Kleene star requires frequent computations on the transposed matrix and powers of the matrix. Consequently, this research focuses particularly on optimizations regarding the following two aspects.

- Computation per block: each block includes four 'float' variables.
- Efficient computation of the transposed matrix and multiplication of two matrices.

Hereafter, we outline the optimization in the implementation.

A. Internal structure of vectors and matrices

For vector data, we prepare a one-dimensional array consisting of the minimum number of blocks required to store the data. Since each block includes four 'float' type variables, the number of required blocks for storing vector $\boldsymbol{a} \in \mathcal{D}^n$ is:

$$b_n = 1 + \mathrm{int}[(n-1)/4]. \tag{6}$$

Fig. 2 depicts the case for $n = 5$. For any unused elements, ε is substituted. For storing matrices, keeping in mind that transposed matrices must be calculated frequently, we prepare a verbose square matrix as the internal structure. Now, consider storing the values of matrix $\boldsymbol{M} \in \mathcal{D}^{m \times n}$ ($m > 1$, $n > 1$). Recalling (6), the number of blocks required to store a $\max(m, n)$ -sized vector is:

$$b_{m,n} = 1 + \mathrm{int}[(\max(m, n) - 1)/4].$$

Noting that $4 \cdot b_{m,n}$ 'float' variables can be stored in this area, we prepare a one-dimensional array where $4 \cdot b_{m,n}$ variables can be stored in both row and column directions. Accordingly, the required number of blocks is $4 \cdot b_{m,n} \cdot b_{m,n}$, as illustrated in Fig. 3.

B. Calculation of transposed matrix

First, we introduce an efficient algorithm for computing the transposed matrix. It is applicable only for a 4×4 -sized matrix, and this makes use of an instruction called 'permutation'. An illustration thereof is presented in Fig.4. Elements shaded gray represent the cells to be moved to other locations with the permutation instruction. In view of this, we propose computing the transposed matrix by executing two steps as shown in Fig. 5. First, the original matrix is divided into block matrices. Then, blocks (i, j), (j, i) are swapped for all blocks $(1 \le i < j \le b_{m,n})$, ignoring the diagonal block matrices. In the second step, transposed matrices of all block matrices are computed for all i and j $(1 \le i \le b_{m,n}, 1 \le j \le b_{m,n})$.

C. Multiplication

According to (1), to obtain the (i, j) th element of $\boldsymbol{X} \otimes \boldsymbol{Z}$, an inner product of the i th row vector of $\boldsymbol{X}$ and the j th column vector of $\boldsymbol{Z}$ must be calculated. However, as shown in Fig. 3, we can perform operations on rows very quickly whereas operations on elements in a vertical direction are not so easy. Hence, this research first computes the transposed matrix of $\boldsymbol{Z}$, then adds the corresponding elements of the i th row vector of $\boldsymbol{X}$ and the j th row vector of $\boldsymbol{Z}^T$, and finally finds the maximum value of these.

D. Kleene star

We explain algorithms for calculating the Kleene star for a given weighted adjacency matrix $\boldsymbol{X} \in \mathcal{D}^{m \times n}$.

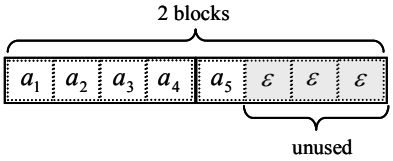

Figure 2. Internal structure of vector-type variables.

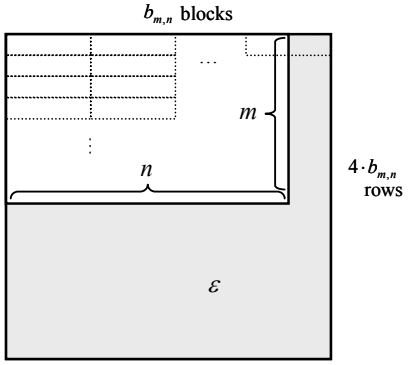

Figure 3. Internal structure of matrix-type variables.

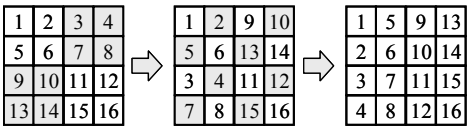

Figure 4. Computation of the transposed matrix of a 4*4-sized matrix.

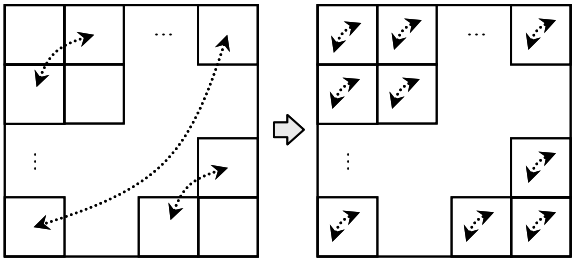

Figure 5. Calculation of the transposed matrix.

In a simple method based on (3), we first prepare two working matrices $\boldsymbol{Z}^{(0)}, \boldsymbol{W}^{(0)} \in \mathcal{D}^{n \times n}$, initialized to $\boldsymbol{e}$, where the upper suffix (*) represents the number of updates with 0 representing the initial value. The values of $\boldsymbol{W}$ are updated in the following manner:

$$\boldsymbol{W}^{(i)} = \boldsymbol{W}^{(i-1)} \otimes \boldsymbol{X}. \tag{7}$$

Subsequently, the values of $\boldsymbol{Z}$ are updated as follows:

$$\boldsymbol{Z}^{(i)} = \boldsymbol{Z}^{(i-1)} \oplus \boldsymbol{W}^{(i-1)}. \tag{8}$$

Equations (7) and (8) are applied repeatedly until $\boldsymbol{W}^{(i)} = \boldsymbol{\varepsilon}$. At this stage, the values of $\boldsymbol{X}^*$ are stored in $\boldsymbol{Z}$.

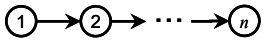

Figure 6. A system with tandem-structured precedence constraints.

Next, the computation algorithms based on methods (a) and (b) are explained. After performing a topological sort on the adjacency matrix $\boldsymbol{X} \in \mathcal{D}^{m \times n}$, both methods update the values of $\boldsymbol{Z}$ based on either (4) or (5). However, we note here that it is better to avoid calculations on rows. Thus, we handle the working matrix based on $\boldsymbol{Z}^T$, and not on $\boldsymbol{Z}$. Then, the update algorithm for method (a) is expressed as:

$$[\boldsymbol{Z}^T]_{ij} \leftarrow [\boldsymbol{Z}^T]_{ij} \oplus \max_{l \in \mathcal{P}(j)} ([\boldsymbol{Z}^T]_{il} + [\boldsymbol{X}]_{jl}).$$

By contrast, the algorithm for method (b) is expressed as :

$$[\boldsymbol{Z}^T]_{ij} \leftarrow [\boldsymbol{Z}^T]_{ij} \oplus \bigoplus_{l=1}^{n} ([\boldsymbol{Z}^T]_{il} + [\boldsymbol{X}]_{jl}).$$

This conversion simplifies the process to be additions of the corresponding elements of the i th row vector of $\boldsymbol{Z}^T$ and the j th row vector of $\boldsymbol{X}$. After all processes have finished, we compute the transposition of the working matrix $\boldsymbol{Z}^T$ to obtain $\boldsymbol{X}^*$.

V. Results of Performance Evaluation

The effect of 'SIMDization' and the performances of the PPE and SPE processors are measured.

Setting the number of nodes to n, consider two cases where the adjacency matrix $\boldsymbol{F}_0$ is (A) dense and (B) sparse. For (A), let all pairs of elements (i, j) that satisfy $i \prec j$ have a precedence constraint $i \rightarrow j$ with probability 1/2. For (B), consider the precedence constraints where all nodes are connected in tandem, as presented in Fig. 6. In both cases (A) and (B), after the initial adjacency matrix has been set, we sort the indexes of the nodes randomly to create experimental adjacency matrices. For the weight matrix $\boldsymbol{P}_k$, we generate the diagonal elements obeying a $[0, 1]$ uniform distribution. The computation times for computing the transition matrix $\boldsymbol{A}_k = (\boldsymbol{P}_k \boldsymbol{F}_0)^* \boldsymbol{P}_k$ from the time that both $\boldsymbol{F}_0$ and $\boldsymbol{P}_k$ are avilable. This is performed for a varying number of nodes, with $n = 10, 20, 40, 80$, using the following two methods; (a) Based on (4): the theoretical time complexity is $O(n \cdot (n+m))$, and (b) Based on (5): the theoretical time complexity is $O(n^3)$.

Since the main objective to estimate the effect of SIMDization, we measure the execution times only when using a single PPE and SPE. The maximum size $n = 80$ is bounded by the size of the LS, since the required storage including all temporary areas is about 256KB for $n = 80$.

The execution environment is Sony Playstation III, Fedora Core 10. We used gcc-4.1.1 and CELL SDK (Software Development Kit) [8] Version 3.1 as complier and library, respectively. We used only a single SPE and PPE, and made experiments for both with SIMD and without SIMD. The

various experimental cases and the corresponding compilation options are summarized in Table 1.

The experiment is performed for cases (1)—(4). In each case, the execution time is measured for hundred trials with a different set of adjacency and weight matrices, and the average computation time is then recorded. The time is measured using the 'gettimeofday()' function. Tables 2—5 show the results, with all times expressed in microseconds. Tables 2 and 3 present the results for dense adjacency matrices.

First, let us consider Table 2. Comparing cases (1) and (2), the effect of SIMDization using an SPE is a speedup of about 2.0 times for $n = 20$ and about 2.8 times for $n = 80$. Comparing cases (3) and (4), the effect of SIMDization using a PPE is a speedup of about 3.0 times for $n = 20$ and about 3.6 times for $n = 80$.

Next, we inspect the results in Table 3. Since method (b) does not create an adjacency list, the effect of SIMDization may be more significant compared with method (a). In fact, comparing cases (1) and (2), the effect of SIMDization is a speedup of about 2.7 times when $n = 20$ and about 4.1 times when $n = 80$. In addition, comparing cases (3) and (4), the speedup is about 5.3 times with $n = 20$ and about 7.1 times with $n = 80$.

Comparing SPEs and PPEs, the absolute computation times using an SPE for $n \geq 20$ are smaller than those using a PPE. Since it is often said that SPEs are more suitable for calculation than a PPE, the results might be consistent with the common belief. However, in SPEs, even scalar variables must be calculated using 128 bit registers, the advantage may not be remarkable if we use many scalar control variables. In fact, for $n = 10$, the computation times using a SPE are greater than using a PPE. This may be due to the above feature. In addition, we should note that the effect of SIMDization using a SPE is less significant, which may also due to the same reason.

Comparing methods (a) and (b), it appears that the difference in execution times is not significant when $n = 10, 20$. However, as n increases, the execution time for method (a) decreases when SIMD instructions are not used. In contrast, the execution time based for method (b) is smaller if we apply SIMD instructions. These results seem to imply that method (b) is suitable for processors in which vector operations are available.

With respect to Tables 4 and 5, the overall ratio of the ratio of timings for SPEs and PPEs and the effect of SIMDization on calculation times are roughly the same as discussed for Tables 2 and 3. However, since these cases focus on sparse adjacency matrices, the time spent on creating the adjacency list is relatively short. This makes the adjacency-list-based method (a) more advantageous than method (b). By contrast, focusing on Tables 2 and 3 again, if the adjacency matrix is dense, it would be better to avoid utilizing an adjacency list. This is confirmed by the fact that the cases based on the adjacency-matrix, method (b), are computed faster than those based on method (a).

As indicated by these results, case (1), using an SPE with SIMD instructions, is the fastest. However, method (a) or method (b) is better, is dependent on the density of the adjacency matrix.

TABLE I. COMPILATION OPTIONS.

Case	Processor	SIMD	Option
(1)	SPE	YES	-O3 -m64 –maltivec -mabi=altivec
(2)		NO	
(3)	PPE	YES	-O3
(4)		NO	

TABLE II. CALCULATION TIMES FOR DENSE MATRICES BASED ON METHOD (A).

n	10	20	40	80
(1)	57	148	763	5,293
(2)	75	296	1,946	14,703
(3)	35	201	1,372	10,261
(4)	82	611	4,700	37,164

TABLE III. CALCULATION TIMES FOR DENSE MATRICES BASED ON METHOD (B).

n	10	20	40	80
(1)	56	133	670	4,592
(2)	83	358	2,454	18,792
(3)	28	146	948	6,780
(4)	103	781	6,081	48,239

TABLE IV. CALCULATION TIMES FOR SPARSE MATRICES BASED ON METHOD (A).

n	10	20	40	80
(1)	56	124	549	3,429
(2)	73	268	1,677	12,355
(3)	28	132	769	5,067
(4)	73	518	3,857	29,749

TABLE V. CALCULATION TIMES FOR SPARSE MATRICES BASED ON METHOD (B).

n	10	20	40	80
(1)	56	135	676	4,620
(2)	81	349	2,362	17,978
(3)	29	147	954	6,803
(4)	99	728	5,608	44,017

VI. CONCLUSION

In this research, a CELL/B.E. processor has been used to examine the effect of a fast computation method for a transition matrix. We have focused on repetitive discrete event systems whose precedence constraints regarding the execution of jobs can be expressed as a DAG. The absolute computation times using the SPE are smaller than those using a PPE. However, it has been proved that the effect of SIMDization is more significant when using a PPE. Comparing the two methods (a) and (b), where the former is based on an adjacency list and the latter on an adjacency matrix, method (a) is faster if the adjacency matrix is sparse, while method (b) is faster if the matrix is dense.

Note that only a single SPE is used in this research whereas six SPEs are available in Linux on the Playstation3. If all these processors are used in parallel, the speedup effect would be much more significant. The implementation and examination thereof remain our future work.

REFERENCES

[1] D. Pham, S. Asano, M. Bolliger, M. Day, and H. Hofstee, "The design and implementation of a first-generation CELL processor," *ISSCC Digital Technical Papers*, pp. 184–185, 2005.

[2] M. Scarpino, *Programming the Cell Processor: For Games, Graphics, and Computation*. New York: Prentice Hall, 2008.

[3] G. Schullerus, V. Krebs, B. Schutter, and T. Boom, "Input signal design for identification of max-plus-linear-systems," *Automatica*, vol. 42, no. 6, pp. 937–943, 2006.

[4] A. Moh, M. Manier, H. Manier, and A. Moudni, "A max-plus algebra modeling for a public transport system., "*Cybernetics and Systems*," vol. 36, pp. 1–16, 2005.

[5] F. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat, *Synchronization and Linearity.* New York: John Wiley & Sons, 1992. [Online]: Available: http://maxplus.org

[6] B. Heidergott, G. J. Olsder, and L. Woude, *Max Plus at Work: Modeling and Analysis of Synchronized Systems.* New Jersey: Princeton University Pr., 2006.

[7] H. Goto, "Efficient calculation of the transition matrix in a max-plus linear state-space representation, "*IEICE Transactions on Fundamentals*, vol. E91-A, no. 5, pp. 1278–1282, 2008.

[8] IBM, *CBE Programmer's Guide version 3.1.* IBM, 2008.

[9] T. Cormen and C. Leiserson, *Introduction to Algorithms.* Massachusetts: MIT Press, 2001.